A Deamortization Approach for Dynamic Spanner and Dynamic Maximal Matching

AARON BERNSTEIN, Rutgers University
SEBASTIAN FORSTER, University of Salzburg, Austria
MONIKA HENZINGER, University of Vienna, Austria

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 article, 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 α if for *every* update σ, the expected time to process σ is at most α. 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 article, we present a *black-box* reduction that converts any data structure with worst-case

A preliminary version of this article was presented at the 30th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2019).
The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP/2007-2013)/ERC Grant Agreement no. 340506. The first author conducted this research while funded by NSF grant 1942010.
Authors' addresses: A. Bernstein, Rutgers University, Department of Computer Science, 110 Frelinghuysen Road, Piscataway, NJ, 08854-8019; email: aaron.bernstein@rutgers.edu; S. Forster, University of Salzburg, Department of Computer Sciences, Jakob- Haringer-Straße 2, 5020, Salzburg, Austria; email: sebastian.forster@plus.ac.at; M. Henzinger, University of Vienna, Faculty of Computer Science, Währinger Straße 29, 1090, Wien, Austria; email: monika.henzinger@univie.ac.at.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2021 Association for Computing Machinery.
1549-6325/2021/10-ART29 $15.00
https://doi.org/10.1145/3469833

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 $O(\log^2(n))$.

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.

CCS Concepts: • **Theory of computation → Dynamic graph algorithms;**

Additional Key Words and Phrases: Spanners, maximal matching

ACM Reference format:
Aaron Bernstein, Sebastian Forster, and Monika Henzinger. 2021. A Deamortization Approach for Dynamic Spanner and Dynamic Maximal Matching. *ACM Trans. Algorithms* 17, 4, Article 29 (October 2021), 51 pages.
https://doi.org/10.1145/3469833

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 article 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 that provide randomized data structures with worst-case guarantees (see, e.g., References [2–4, 16, 18, 25, 31, 33, 36]), often via a complex "deamortization" of previous results.

In this article, 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 α if for *every* update σ, the expected time to process σ is at most α. On its 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* α if the maximum update time of any type of update (e.g., insertion or deletion) is α.

Theorem 1.1. *Let A be an algorithm that maintains a dynamic data structure D with worst-case expected update time α 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 $\sigma_1, \sigma_2, \ldots, A'$ processes each update σ_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 $D_1, D_2, \ldots, D_{\Theta(\log(n))}$, 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. However, combining A' with the very recent black-box reduction in Reference [38], we can turn A' 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 $O(\log^6(n) + 1/\epsilon)$. (See the end of Section 4 for more explanations.)

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 α 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 α 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 [6], and this tradeoff is presumed tight under Erdős's girth conjecture. The dynamic spanner problem was introduced by Ausiello, Franciosa, and Italiano [5] and has been improved upon by References [8, 16, 21]. There currently exist near-optimal amortized expected bounds: A $(2k-1)$-spanner can be maintained with expected amortized update time $O(1)^k$ [8] or time $O(k^2 \log^2(n))$ [24]. 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 [16]; no $o(n)$ worst-case update time was known for larger k. All of these algorithms exhibit the stretch/space tradeoff mentioned above, up to polylogarithmic factors in the size of the spanner[1].

We give the first dynamic spanner algorithm with polylog worst-case update time with high probability for any constant k, which significantly improves upon the result of Reference [16] both in update time and in range of k. Our starting point is the earlier result of Baswana, Khurana, and Sarkar [8] 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 [37]. Due to conditional lower bounds of $\Omega(\sqrt{m})$ on the time per operation for this problem [1, 30], there is a large body of work on the dynamic *approximate* matching problem [4, 7, 11, 13–15, 18, 27, 28, 34, 35, 39]. 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

[1]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 $\sigma_1, \sigma_2, \ldots$ *before* the dynamic algorithm starts to operate, and the random choices of the algorithm then define a distribution on the time to process each σ_i. This is called an *oblivious* adversary. Our dynamic algorithms for spanners and matching share this assumption, as does all work prior to the conference version of this article. A randomized dynamic matching algorithm that does not need this assumption was presented in Reference [40].

a well-studied object in its own right (see, e.g., References [7, 23, 26, 29, 32, 34, 39]). The groundbreaking result of Baswana, Gupta, and Sen [7] showed how to maintain a maximal matching (and so 2-approximation) with $O(\log(n))$ expected amortized update time. Solomon [39] improved the update time to $O(1)$ expected amortized. There has been recent work on either deamortizing or derandomizing this result [4, 11, 14, 15, 18]. Most notably, the two independent results in References [18] and [4] both present algorithms with polylog high-probability worst-case update time that maintain a $(2+\epsilon)$-approximate matching. Unfortunately, all these results come 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 Reference [4]. Much like for dynamic spanner, we use an existing amortized algorithm as our starting point: namely, the $O(\log(n))$ amortized algorithm of Reference [7]. 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. [7]: 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 that 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. Our tools for proving worst-case expected guarantees can be used to de-amortize other existing dynamic algorithms has already been used for two novel fully dynamic maximal independent set algorithms [9, 19] and we expect it to find further use. For example, the dynamic coloring algorithm of Reference [12], the dynamic spectral sparsifier algorithm of Reference [3], the dynamic distributed maximal independent set algorithm of Reference [17], and the dynamic distributed spanner algorithm of Reference [8] (all amortized) seem like natural candidates for our approach.

Section 2 provides a proof of the black-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 α, 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 ℓ be a parameter for the length of the update sequence to be considered that is polynomial in n. Then there exists an algorithm A' with the following properties:*

1. *For any sequence of updates $\sigma_1, \sigma_2, \ldots, \sigma_\ell$ with ℓ polynomial in n, A' processes each update σ_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 $D_1, D_2, \ldots, D_{\Theta(\log(n))}$, 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 versions of the algorithm A, denoted $A_1, \ldots, A_q$, 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, then D_i is marked as *fixed*, otherwise it is marked 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 σ_j the algorithm adds σ_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, then A_i marks itself as fixed; otherwise, it marks itself as broken. If at the end of update σ_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 $rq = O(\alpha \log^2(n))$ 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 α (for average), r (for reduction), and ℓ (for length)* as follows: Given a finite sequence of possibly dependent random variables $X_1, X_2, \ldots, X_\ell$ such that for each t, $E[X_t] \leq \alpha$, we define a sequence of counters C_t that changes over a finite sequence of time steps. Let $C_0 = 0$ and let $C_t = \max(X_t + C_{t-1} - r, 0)$.

As we show in Lemma 2.3 with constant probability C_t is 0. We use this fact as follows: Let A_i be any version. 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 σ_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 jth 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 $(1/2)^q = 1/n^c$. □

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

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,

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

We now need to bound $\mathbf{Pr}[B_i]$. 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 $\mathbf{Pr}[E_1] \leq \mathbf{Pr}[E_2]$, we have that

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

Now observe that, by linearity of expectation,

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

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

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 α (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 ℓ, 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 $\mathbf{Pr}[C_t = 0]$ is very small.

To exhibit this bad case, consider the following sequence $X_1, X_2, \ldots X_\ell$, 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 $\mathbf{Pr}[C_\ell = 0]$? For each $t \leq \ell$ if $X_t \neq 0$, the counter will reduce by $r(\ell + 1 - t)$ from time t to time ℓ, 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, $\mathbf{Pr}[C_\ell > 0] = \mathbf{Pr}[Y_1 \vee Y_2 \ldots \vee Y_\ell]$. This probability is hard to bound exactly, but note that since the Y_t's are independent random variables between 0 and 1 and we can apply the following Chernoff bound:

Lemma 2.4 (Chernoff Bound). *Let $Y_1, Y_2, \ldots, Y_k$ 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 $\delta > 0$:

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

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

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

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

Thus, to have $\mathbf{Pr}[C_\ell = 0] \geq 1/2$, 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 $r = \Omega(\alpha \log(\ell))$.

This now completes the proof of all parts of Theorem 2.1.

Finally, we observe that the restriction to an update sequence of finite length is mainly a technical constraint and we show next how to remove it. The basic idea is to periodically rebuild a new copy of the data structure "in the background" by spreading this computation over the time period.

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 $D_1, D_2, \ldots, D_{\log(1/p)}$ of the active instance, where p is a suitably chosen parameter. 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. □

3 DYNAMIC SPANNER WITH WORST-CASE EXPECTED UPDATE TIME

In this section, we give a dynamic spanner algorithm 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. [8] and replace a crucial subroutine requiring deterministic amortization by a randomized counterpart with worst-case expected update time guarantee. In Section 3.1, we first give a high-level overview explaining where the approach of Baswana et al. [8] requires (deterministic) amortization and how we circumvent it. We then, in Section 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 Section 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, then 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 $p = \frac{1}{n^{1/k}}$. 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 clusters. 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 (after possibly being extended by the hooks) 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.

The main idea why this hierarchy gives a good spanner is the following: If a vertex belonging to an unsampled cluster has many neighboring clusters, then one of them is likely to be a sampled one and so the vertex joins a sampled cluster and is passed on to the next level of the hierarchy. Conversely, if it stays at the current level of the hierarchy, then it only has few neighboring clusters, namely, $O(\frac{1}{p}) = O(n^{1/k})$ many in expectation. For such vertices, one can therefore afford to add one edge per neighboring cluster to the spanner. By doing so, it is ensured that there is a path of length $2k-1$ for each incident edge as every cluster has radius at most $k-1$.

This hierarchy is maintained with the help of sophisticated data structures and some crucial applications of randomization to keep the expected update time low. One important aspect for bounding the update time in such a hierarchical approach is the following: It is not sufficient to analyze the update time at each level of the hierarchy in isolation, as updates performed to one level might lead to changes in the clustering that lead to *induced updates* to the next level. In principle, by such a propagation of updates, a single update to the input graph might lead to an exponential number of induced updates to be processed by the last level. 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 (both the choice of the sampled singleton clusters and the choice of the hooks), then every edge incident on u has the same probability of being the hook of u, i.e., the probability of e being the hook of u is $\frac{1}{\deg(u)}$. Thus, the expected update time is $\frac{1}{\deg(u)} \cdot O(\deg(u)) = O(1)$.

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, then it would join one of the single-edge clusters with high-probability. As there are ℓ edges incident on u from these single-edge clusters, this gives a probability of approximately $\frac{1}{\ell}$ for some deleted edge (u, v) being the hook, which can be much larger than $\frac{1}{\deg_i(u)}$. This problem would not appear if among all edges going to neighboring clusters a pth fraction would be incident on sampled clusters. Then, intuitively speaking, one could argue that the probability of some edge $e = (u, v)$ being the hook of u is at most $p \cdot \frac{1}{\Omega(p \deg_i(u))}$, the probability that the cluster containing v 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 pth fraction goes 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 $\Theta(\frac{1}{p}\log(n))$ clusters), a standard Chernoff bound for binary random variables guarantees that a pth 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 pth 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. 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 $0 \le j \le O(\log(n))$) exceeds a bucket-specific threshold of α_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 $O(1)^{k/2}$ as in the case of Baswana et al. By a more careful analysis, we do actually obtain the $O(1)^{k/2}$-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 $\sum_{t\ge 1} 2^t/e^{(2^t)}$, 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.

In the following, we review the algorithm of Baswana et al. [8] 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 Static Spanner Construction. Let us first explain the principle behind the algorithm of Baswana et al. by reviewing a purely static version of the construction.

Given an integer parameter $k \geq 2$, the construction uses clusterings $C_0, C_1, \ldots, C_{k-1}$ of subgraphs $G_0 = (V_0, E_0), G_1 = (V_1, E_1), \ldots, G_{k-1} = (V_{k-1}, E_{k-1})$, both to be specified in the following, 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 partially determined by a hierarchy of randomly sampled subsets of vertices $S_0 \supseteq S_1 \supseteq \cdots \supseteq S_k$ in the sense that each cluster c in C_i contains a designated vertex of S_i called the *center* of c. This sampling is performed by setting $S_0 = V$, $S_k = \emptyset$, and by forming S_i, for each $1 \leq i \leq k-1$, by selecting each vertex from S_{i-1} independently with probability $p = \frac{1}{n^{1/k}}$. In addition to the clusterings, the construction uses a forest (V_i, 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. Informally, level i of this hierarchy denotes all the sets of the construction indexed with i. Initially, $G_0 = G$, $F_0 = \emptyset$ 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 set of edges F_{i+1}, based on the graph $G_i = (V_i, E_i)$, the clustering C_i, the edge set F_i, and the set of vertices 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 of 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 any other vertex of 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 construction takes an arbitrary edge $(u, v) \in \bigcup_{c \in R_i} E_i(u, c)$ as the *hook* of u at level i, called hook(u, 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 $c \in C_i \setminus R_i$. Finally, 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 at least one of u and v belongs to a sampled cluster (in R_i) at level i.

The static spanner H now consists of the set of edges $\bigcup_{0 \leq i \leq k-1}(F_i \cup X_i)$. To analyze the stretch of H, consider some edge $e = (u, v)$ and let i be the largest index such that $e \in E_i$. If u and v are contained in the same cluster in the clustering C_i, then the path from u to v in F_i via the common cluster center has length at most $2i \leq 2k-2$, as each cluster has radius at most $i \leq k-1$. If u and v are contained in different clusters in the clustering C_i, then X_i contains an edge $e' = (u, v')$ from u to the cluster of v. Now there is a path in H of length at most $2i+1 \leq 2k-1$ from u to v' by first taking the edge e' to v' and then taking the path from v' to v in F_i via the common cluster center.

To analyze the size of the spanner, observe first that each forest F_i consist of at most $n-1$ edges. Furthermore, for each $0 \leq i \leq k-2$, each vertex in $V_i \setminus V_{i+1}$, which is the set of vertices not being adjacent to a sampled cluster, is adjacent to at most $\frac{1}{p} = n^{1/k}$ clusters in expectation (all of which are non-sampled clusters) Thus, the number of edges contributed to X_i by each vertex in $V_i \setminus V_{i+1}$ is at most $n^{1/k}$ in expectation. At level $k-1$, no clusters are sampled ones anymore and X_{k-1} contains for each vertex in V_{k-1} one edge to each neighboring cluster. As the number of clusters has been reduced to $n^{1/k}$ in expectation at level $k-1$, each vertex in V_{k-1} again contributes $n^{1/k}$ edges to X_{k-1} in expectation. This results in an overall spanner size of $O(kn^{1+1/k})$ in expectation.

3.2.2 Dynamic Spanner Maintenance. The dynamic spanner algorithm uses the same definitions as above, with some minor modifications regarding how the hooks and the sets E_i are determined, and an additional edge set Y_i being included in H for each level i. Note that the sampling of $S_0 \supseteq S_1 \supseteq \cdots \supseteq S_k$ is performed a priori at the initialization and does not change over the course of the algorithm. At each level i (for $0 \leq i \leq k-1$), instead of selecting an arbitrary edge from $(u,v) \in \bigcup_{c \in R_i} E_i(u,c)$ as the hook of u for each vertex $u \in \mathcal{N}_i$, the hook is picked uniformly at random guaranteeing the following "hook invariant":

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

The main idea of Baswana et al. is that this simple method of choosing the hook leads to a fast update time if an additional filtering step is performed for selecting the edges that go to the next level.

For this purpose, the algorithm maintains, for each $u \in \mathcal{N}_i$, and 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[2]:

(F1) For every $0 \leq j \leq \lfloor \log_g(n) \rfloor$ 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 $\mathcal{I}_i(u)$.[3]

Intuitively, the set $\mathcal{F}_i(u)$ is a *filter* on the edges from u to non-sampled neighboring clusters and only edges to non-sampled clusters in $\mathcal{F}_i(u)$ may 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 to compensate for these missing edges to keep the spanner intact. For this purpose, the algorithm maintains a set of edges Y_i 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.[4] 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 ϵ and a.

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:

[2]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. We further deviate by suggesting to maintain this partitioning into buckets (which we call dynamic filtering) for each node in V_i (a superset of $\mathcal{N}_i$). This does not increase the asymptotic running time of the overall algorithm and avoids special treatment when vertices join or leave $\mathcal{N}_i$. Baswana et al. explicitly provide an argument for charging the initialization for of a vertex joining $\mathcal{N}_i$ to a sequence of induced updates. We believe that our variant that avoids initialization slightly simplifies the formulation of Theorem 3.3.

[3]The filtering algorithm of Baswana et al. guarantees the following stronger version of (F3): For every cluster $c \in C_i \setminus R_i$ either $E_i(u,c) \subseteq \mathcal{F}_i(u)$ or $c \in \mathcal{I}_i(u)$. However, for the spanner algorithm to be correct, only the weaker guarantee of (F3) stated above is necessary. We will use this degree of freedom in our new filtering algorithm to avoid unnecessary "bookkeeping" work.

[4]Note that the lack of "disjointness" between $\mathcal{F}_i(u)$ and $\mathcal{I}_i(u)$ might lead to the situation that some edge is contained in both Y_i and $\mathcal{F}_i(u)$. This was not the case in the original algorithm of Baswana et al., but it is correct to allow this behavior and allows us to avoid unnecessary "bookkeeping" work in our new filtering algorithm.

- 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 $\mathcal{E}_i(v)$.

Having defined this hierarchy, the dynamic spanner H consists of the set of edges $\bigcup_{0 \le i \le k-1}(F_i \cup X_i \cup Y_i)$.

3.2.3 Sketch of Analysis. As explained above, it follows from standard arguments that $|F_i \cup X_i| \le O(n^{1+1/k})$ for each $0 \le i \le k-1$. Furthermore, the size of Y_i is bounded by $n \cdot \max_u |\mathcal{I}_i(u)|$ for each $0 \le i \le k-1$. 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—by using appropriate data structures—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 ([8]). *For every $0 \le i \le k-1$ and every edge $(u, v) \in E_i$, $\mathbf{Pr}[(u, v) = \text{hook}(u, i)] \le \frac{1+2\epsilon}{|\mathcal{E}_i(u)|}$ for any constant $0 < \epsilon \le \frac{1}{4}$.*

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 ([8]). *Let $o_1, \ldots, o_\ell$ be ℓ positive numbers such that the ratio of the largest to the smallest number is at most b, and let $Z_1, \ldots, Z_\ell$ be ℓ independent random variables such that Z_i takes value o_i with probability p and 0 otherwise. Let $\mathcal{Z} = \sum_{1 \le i \le \ell} Z_i$ and $\mu = \mathbf{E}[\mathcal{Z}] = \sum_{1 \le i \le \ell} o_i p$. There exists a constant $\gamma \le 80$ such that if $\ell \ge \gamma a b \frac{1}{\epsilon^3 p} \ln(n) \log(b)$ for any $0 < \epsilon \le \frac{1}{4}$, $a > 1$, and a positive integer n, then the following inequality holds:*

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

The running-time argument sketched above only bounds the running time of each level "in isolation." For every $0 \le i \le k-1$, 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 [8]). *Assume that for constant $0 < \epsilon < 1$ and $a > 1$ there is a fully dynamic edge filtering algorithm $\mathfrak{F}$, 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 $k \ge 2$, 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 $\mathfrak{S}$.*

3.2.4 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, i.e., there will be a separate instance of the filtering algorithm for each vertex in V_i. The algorithm takes parameters $0 < \epsilon < 1$ and $a > 1$ and fixes some choice of $\lambda \geq g > 1$. It operates on the subset of edges of E_i incident on u and any vertex v in a non-sampled cluster $c \in C_i \setminus R_i$. 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.[5] 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 numbered from 0 to $\lfloor \log_g(n) \rfloor$, 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 c 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, the ratio between the largest and the smallest value of $|E_i(u, c)|$ among clusters c in the same bucket is at most λ^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 ℓ comes from Theorem 3.2; a is a constant that controls the error probability, ϵ controls the amount of deviation from the mean in the Chernoff bound, and γ 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 Reference [8]). 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 λ that has small update time. An additional goal in the design of 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 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 to have a dynamic filtering algorithm that only performs $O(1)$ changes to $\mathcal{F}_i(u)$ per update to its input.

3.2.5 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 [8]). *For any $a > 1$ and any $0 < \epsilon \leq \frac{1}{4}$, there is a dynamic filtering algorithm with amortized update time $O(\frac{1}{\epsilon})$ for which the amortized number of changes performed to $\mathcal{F}_i(u)$ 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 $T(n) = O(\frac{1}{\epsilon})$.*

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:

[5] Note that if vertices join or leave clusters, then the dynamic filtering algorithm only sees updates for the corresponding edges.

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 ϵ 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 ℓ clusters and all inactive buckets contain at most $\kappa\ell$ clusters for some constant κ. An active bucket will be inactivated if its size falls to ℓ 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 these 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 10.6 changes to $\mathcal{F}_i(u)$ in expectation, i.e., $U(n) = 10.6$ 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 worst-case expected update time $O(14^{k/2}\log(n))$.*

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 $O(14^{k/2}\log^3(n))$ 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 away from $\mathcal{F}_i(u)$. We can also, somewhat more aggressively, keep away the edges incident from the first ℓ clusters of large buckets out of $\mathcal{F}_i(u)$. 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 deterministic based only on the current partitioning of the clusters into buckets and on an arbitrary, but fixed ordering of vertices, clusters, and edges. This is a bit similar to the idea in Reference [16] of always keeping the "first" few incident edges of each vertex in the spanner.

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 $O(1/2^{e^t})$. 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. [15]. However, such a deamortization needs to solve non-trivial challenges and the other parts of the dynamic spanner algorithm would still be randomized. Furthermore, we believe that the probabilistic threshold technique leads to a significantly 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 $0 \leq j \leq \lfloor \log(n) \rfloor$, 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 $0 \leq j \leq \lfloor \log(n) \rfloor$, 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.

We set $\lambda = 2^{\lceil \log(4+\ln(n)) \rceil} = O(\log(n))$, $\ell = 4\gamma a \lambda^2 \frac{1}{\epsilon^3} n^{1/k} \ln(n) \ln(\lambda) = O(n^{1/k} \log^3(n) \log\log(n))$ and, for every $0 \leq j \leq \lfloor \log(n) \rfloor$, we set $\alpha_j = 2^j$. 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 $\mathcal{I}_i(u)$.

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 ℓ 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, where we denote the unique cluster of v by c:

- If the edge (u, v) was inserted, then check if one of the following cases applies:
  - —If $|E_i(u, c)| = 1$ after the insertion (i.e., c becomes a neighbor of u), then move c into bucket 0 by performing the following steps:
    (1) Add c to B_0.
    (2) Add (u, v) to T_0.
  - —If $|E_i(u, c)| \geq 2\alpha_j$ after the insertion, where j is the number c's current bucket, then 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, then check if one of the following cases applies:
  - —If $|E_i(u, c)| = 0$ after the deletion (i.e., c ceases to be a neighbor of u), then move c out of bucket 0 by performing the following steps:
    (1) Remove c from B_0.
    (2) Remove (u, v) from T_0.
  - —If $|E_i(u, c)| \leq \frac{\alpha_j}{2}$ after the deletion, where j is the number c's current bucket, then 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 $|E_i(u, c)| \leq \frac{\alpha_j}{2^t}$ (i.e., $t = \lfloor \log(\frac{\alpha_j}{|E_i(u,c)|}) \rfloor$). 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. 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 update time and the number of changes to $\mathcal{F}_i(u)$ per update to some $E_i(u, c)$ for some cluster c. These changes are also called induced updates.

If by the update c becomes a neighbor of u, then one cluster is added to B_0 and one edge is added to T_0. Similarly, if by the update c ceases to be a neighbor u, then one cluster is removed from B_0 and one edge is removed from T_0. Clearly, both of these cases lead to at most 2 changes to $\mathcal{F}_i(u)$ and a running time of $O(\log n)$ as both B_0 and T_0 are organized as binary search trees.

Now observe that each other type of update causes at most one move of c from some bucket j to some other bucket j'. Each such move can be processed in time $O(|E_i(u,c)| \log n)$, as only the cluster c is moved from some binary search tree B_j to another binary search tree $B_{j'}$ and correspondingly only $|E_i(u,c)|$ cluster-edges pairs are moved from the binary search tree T_j to the binary search tree $T_{j'}$. To analyze the number of changes to $\mathcal{F}_i(u)$, consider the following case distinction for some cluster-edge pair $(c,(u,v))$ being moved from T_j to $T_{j'}$:

- If $(c,(u,v))$ is among the first $\ell \cdot \lambda\alpha_j$ cluster-edge pairs of T_j before being removed from T_j and is among the first $\ell \cdot \lambda\alpha_{j'}$ cluster-edge pairs of $T_{j'}$ after being added to $T_{j'}$, then (u,v) is neither contained in $\mathcal{F}_i(u)$ before nor after the move. Furthermore, at most one cluster-edge pair might start being among the first $\ell \cdot \lambda\alpha_j$ pairs in T_j (resulting in the removal of the corresponding edge from $\mathcal{F}_i(u)$) and at most one cluster-edge pair might stop being among the first $\ell \cdot \lambda\alpha_{j'}$ pairs in $T_{j'}$ (resulting in the addition of the corresponding edge to $\mathcal{F}_i(u)$). Thus, we perform at most 2 changes to $\mathcal{F}_i(u)$ for moving $(c,(u,v))$.
- If $(c,(u,v))$ is among the first $\ell \cdot \lambda\alpha_j$ cluster-edge pairs of T_j before being removed from T_j and is not among the first $\ell \cdot \lambda\alpha_{j'}$ cluster-edge pairs of $T_{j'}$ after being added to $T_{j'}$, then (u,v) is not contained in $\mathcal{F}_i(u)$ before the move, but it is contained in $\mathcal{F}_i(u)$ after the move. Furthermore, at most one cluster-edge pair might start being among the first $\ell \cdot \lambda\alpha_j$ pairs in T_j (resulting in the removal of the corresponding edge from $\mathcal{F}_i(u)$) and no cluster-edge pair will stop being among the first $\ell \cdot \lambda\alpha_{j'}$ pairs in $T_{j'}$. Thus, we perform at most 2 changes to $\mathcal{F}_i(u)$ for moving $(c,(u,v))$.
- If $(c,(u,v))$ is not among the first $\ell \cdot \lambda\alpha_j$ cluster-edge pairs of T_j before being removed from T_j and is among the first $\ell \cdot \lambda\alpha_{j'}$ cluster-edge pairs of $T_{j'}$ after being added to $T_{j'}$, then (u,v) is contained in $\mathcal{F}_i(u)$ before the move, but it is not contained in $\mathcal{F}_i(u)$ anymore after the move. Furthermore, no cluster-edge pair will start being among the first $\ell \cdot \lambda\alpha_j$ pairs in T_j and at most one cluster-edge pair might stop being among the first $\ell \cdot \lambda\alpha_{j'}$ pairs in $T_{j'}$ (resulting in the addition of the corresponding edge to $\mathcal{F}_i(u)$). Thus, we perform at most 2 changes to $\mathcal{F}_i(u)$ for moving $(c,(u,v))$.
- If $(c,(u,v))$ is not among the first $\ell \cdot \lambda\alpha_j$ cluster-edge pairs of T_j before being removed from T_j and is not among the first $\ell \cdot \lambda\alpha_{j'}$ cluster-edge pairs of $T_{j'}$ after being added to $T_{j'}$, then (u,v) is contained in $\mathcal{F}_i(u)$ before and after the move. Furthermore, no cluster-edge pair will start being among the first $\ell \cdot \lambda\alpha_j$ pairs in T_j and no cluster-edge pair will stop being among the first $\ell \cdot \lambda\alpha_{j'}$ pairs in $T_{j'}$. Thus, we perform no changes to $\mathcal{F}_i(u)$ for moving $(c,(u,v))$.

Thus, for each move of a cluster c, we incur at most $2|E_i(u,c)|$ 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 $1 \le t \le \log(\lambda) - 1$, let p_t be the probability that $2^t\alpha_j \le |E_i(u,c)| < 2^{t+1}\alpha_j$ 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 $2|E_i(u,c)|$ updates, where $|E_i(u,c)| < 2^{t+1}\alpha_j$ with probability p_t and $|E_i(u,c)| \le n$ 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

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

We now bound p_t, the probability that $2^t\alpha_j \le |E_i(u,c)| < 2^{t+1}\alpha_j$ 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 $p_t \le \frac{1}{\alpha_j}$. For $2 \le t \le \log(\lambda) - 1$, 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

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

Here, we use the inequality $(1 - \frac{1}{x})^x \le \frac{1}{e}$, 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

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

We can now bound the expected number of induced updates by

$$\begin{aligned}
\sum_{1\le t\le \log(\lambda)-1} p_t \cdot 2 \cdot 2^{t+1}\alpha_j + q \cdot 2n &= \frac{1}{\alpha_j} \cdot 2 \cdot 2^2\alpha_j + \sum_{2\le t\le \log(\lambda)-1} \frac{1}{\alpha_j} \cdot \frac{1}{e^{2^t-2}} \cdot 2 \cdot 2^{t+1}\alpha_j + \frac{1}{e^2 n} \cdot 2n \\
&\le 8 + 4 \cdot \sum_{2\le t<\infty} \frac{2^t}{e^{2^t-2}} + 0.28 \\
&\le 8 + 4 \cdot 0.57 + 0.28 \\
&\le 10.6 .
\end{aligned}$$

Moving Down. For every $1 \le t \le \log(\lambda) - 1$, let p_t be the probability that $\frac{\alpha_j}{2^{t+1}} < |E_i(u,c)| \le \frac{\alpha_j}{2^t}$ 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 $2|E_i(u,c)|$ 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

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

We now bound p_t, the probability that $\frac{\alpha_j}{2^{t+1}} < |E_i(u,c)| \le \frac{\alpha_j}{2^t}$ when c is moved down. For $t = 1$, we clearly have $p_1 \le \frac{2^3}{\alpha_j} = \frac{8}{\alpha_j}$, as this is the probability that just a single coin flip made the cluster move down. For $2 \le t \le \log(\lambda) - 1$, observe that for $|E_i(u,c)| \le \frac{\alpha_j}{2^t}$ to hold, there must have been at least $t - 1$ subsequences of deletions such that after every deletion in subsequence s we had $\frac{\alpha_j}{2^{s+1}} < |E_i(u,c)| \le \frac{\alpha_j}{2^s}$ (where $1 \le s \le t - 1$). Observe that the sth subsequence consists of $d_s := \frac{\alpha_j}{2^s} - \frac{\alpha_j}{2^{s+1}} = \frac{\alpha_j}{2^{s+1}}$ many deletions. Remember that during the sth subsequence the probability of c moving down is $r_s := \min(\frac{2^{2s+1}}{\alpha_j}, 1)$. If $r_s < 1$, then by the inequality $(1 - \frac{1}{x})^x \le \frac{1}{e}$ we have

$$(1 - r_s)^{d_s} = \left(1 - \frac{2^{2s+1}}{\alpha_j}\right)^{\frac{\alpha_j}{2^{s+1}}} = \left(1 - \frac{2^{2s+1}}{\alpha_j}\right)^{\frac{\alpha_j}{2^{2s+1}}\cdot 2^s} \le \frac{1}{e^{2^s}}.$$

In the other case, $r_s = 1$, we clearly have $(1-r_s)^{d_s} = 0 \leq \frac{1}{e^{2^s}}$. Now p_t is determined by one "heads" preceded by at least d_s "tails" for each subsequence s, i.e., p_t is bounded by

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

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

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

We can now bound the expected number of induced updates by

$$\begin{aligned} \sum_{1 \leq t \leq \log(\lambda)-1} p_t \cdot 2 \cdot \frac{\alpha_j}{2^t} + q \cdot 2n &= p_1 \cdot 2 \cdot \frac{\alpha_j}{2} + \sum_{2 \leq t \leq \log(\lambda)-1} p_t \cdot 2 \cdot \frac{\alpha_j}{2^t} + q \cdot 2n \\ &\leq \frac{8}{\alpha_j} \cdot 2 \cdot \frac{\alpha_j}{2} + \sum_{2 \leq t \leq \log(\lambda)-1} \frac{2^{2t+1}}{e^{2^t-2}\alpha_j} \cdot 2 \cdot \frac{\alpha_j}{2^t} + \frac{1}{e^2 n} \cdot 2n \\ &\leq 8 + 4 \cdot \sum_{2 \leq t < \infty} \frac{2^t}{e^{2^t-2}} + 0.28 \\ &\leq 8 + 4 \cdot 0.57 + 0.28 \\ &\leq 10.6 . \end{aligned}$$

This concludes the proof that the expected number of induced updates is at most 10.6.

4 DYNAMIC MAXIMAL MATCHING WITH WORST-CASE EXPECTED UPDATE TIME

In this section, we turn to proving Theorem 1.5.[6] We achieve our result by modifying the algorithm of Baswana et al. [7], 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 $\textsc{mate}(v)$, for matched v, to be 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

[6]The correctness proof had to be significantly extended due to a mistake in our SODA 2019 paper.

free and must scan all of their neighbors looking for a new neighbor to match to. The update time is thus $O(\max\{\text{DEGREE}(u), \text{DEGREE}(v)\})$. This is the only expensive operation.

At a very high level, the idea of the Baswana et al. algorithm 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. Let $\text{L}_0 = \lfloor \log_4(n) \rfloor$.

- 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 $\text{L}_0 + 2$ *levels*, numbered from -1 to L_0. 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 in the hierarchy (via what we call a RISE operation) 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.[7] We now describe some guarantees of the Baswana et al. algorithm. Note that the hierarchy only maintains an upper bound on $N_{<j}(v)$ (Invariant 3), *not* a lower bound; a lower bound on $N_{<j}(v)$ only comes into play when v picks a new matching edge (Matching Property). More specifically, right before a mate is randomly selected for a node v on level j the algorithm makes sure that $|N_{<j}(v)| \geq 4^j$

- **Invariant 1:** Each edge is owned by exactly one endpoint, and if the endpoints of the edge are at different levels, then 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 ≥ 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.
- **Invariant 4:** Both endpoints of a matched edge are at the same level.
- **Matching Property:** If a vertex v at level $j > -1$ is (temporarily) unmatched, then 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

[7] Baswana et al. gave an equivalent definition in terms of the O_v and $\mathcal{E}_v^i$ structures.

(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).

Invariants 1 and 3 combined imply that $|O_v| \leq \phi_v(j+1) \leq 4^{\text{LEVEL}(v)+1} = O(4^{\text{LEVEL}(v)})$ and $N_{=\text{LEVEL}(v)}(v) \leq 4^{\text{LEVEL}(v)+1} = O(4^{\text{LEVEL}(v)})$ if $\text{LEVEL}(v) < \text{L}_0$. For $\text{LEVEL}(v) = \text{L}_0$, $4^{\text{L}_0+1} \geq n$, so trivially $|O_v| = O(4^{\text{LEVEL}(v)})$ and $N_{=\text{LEVEL}(v)}(v) = O(4^{\text{LEVEL}(v)})$.

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, the endpoint v is at level ≥ 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 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 Reference [7] for the original algorithm description (and its analysis).

- CheckForRise(v, i) increases $\phi_v(i)$ by one, whereas Decrement-$\phi(v, i)$ decreases it. (The paper of Baswana et al. instead called the increment function Increment-ϕ, but we choose CheckForRise because it better fits the details of our algorithm.) Note that CheckForRise(v, i) might trigger a call to Rise(v, i, j) and the logic that we use for triggering this call differs from that in Reference [7] as we also have probabilistic rises; see below.
- Rise(v, i, j) (new notation) moves a vertex v from level i to level j. This results in changes to many of the 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 $w \in N_{<j}(v)$, and every $i < k \leq j$. A careful analysis bounds the total amount of bookkeeping work at $O(4^j)$ (see Lemma 4.4).
- Fall(v, i) (new notation) moves v from level i to level $i-1$. As above this leads to bookkeeping work: O_w, $\mathcal{E}_w^i$, and $\mathcal{E}_w^{i-1}$ change for many neighbors of w of v. Note that only edges (v, w) previously owned by v are affected, so by Invariant 3, the total amount of bookkeeping work is at most $|O_v| = O(4^i)$.
  The algorithm must also do CheckForRise(w, i) for every w that was previously in $N_{<i}(v)$, incrementing $\phi_w(i)$. 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 is to bound this cascade.
- FixFreeVertex(v) handles the case when a vertex v is unmatched; this can happen because the matching edge incident to v 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 Fall(v, i), followed by FixFreeVertex(v). However, if $|N_{<i}(v)| \geq 4^i$, then v remains at level i and picks a new mate by executing RandomSettle(v, i).
- RandomSettle(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 Rise(w, ℓ, i) (to satisfy Invariant 4) and then matches v to w. Note that

if $\ell \neq -1$, then w had a previous mate w' that is now unmatched, so the algorithm now does 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 CheckForRise(u, j) and CheckForRise(v, j) for every $j > \text{level}(v)$. This takes time $O(\log(n))$ and might additionally result in some level ℓ 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 Rise$(v, \text{level}(v), \ell)$ (or Rise$(u, \text{level}(u), \ell)$). If $\phi_v(\ell) \geq 4^\ell$ for multiple levels ℓ, then v rises to the highest such ℓ.

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 Decrement-ϕ. If (u, v) was *not* a matching edge, then the work ends there: Unlike with CheckForRise, the procedure Decrement-ϕ 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 FixFreeVertex(u), and FixFreeVertex(v).

Analysis Sketch. Whereas our final algorithm is very similar to the original algorithm of Baswana et al., our analysis is mostly 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 CheckForRise(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 Modified Algorithm

Recall the definition of $\phi_v(j)$ for any vertex v with $i = \text{level}(v)$ and level j:

$$\phi_v(j) = \begin{cases} |N_{<j}(v)| & \text{if } j > i \\ 0 & \text{otherwise.} \end{cases}$$

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 that increases $\phi_v(i)$ from $4^i - 1$ to 4^i is guaranteed to lead to the expensive execution of 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 that 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 $\mathbf{Pr}[\text{mate}(v) = u] \leq 1/4^i$, 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 $\mathbf{Pr}[\text{MATE}(v) = u] \leq 1/4^i$. To see this, consider the following star graph with center v. In the sequence of updates to the edges incident 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', $x_1, x_2, \ldots, x_{4^i-1}$ and $y_1, y_2, \ldots, y_{4^i-1}$. 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, $\mathbf{Pr}[\text{MATE}(v) = v'] \to 1$. 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'), then this deletion is near-guaranteed to be expensive because $\mathbf{Pr}[\text{MATE}(v) = v'] \sim 1$.

One way to overcome this issue is to give v a small probability of resetting its matching edge every time a neighbor of v undergoes certain kinds of changes in the hierarchy; this would ensure that even if (v, v') becomes the matching edge at some point during the process, it will not stick forever. This is the approach we will take.

4.2.1 List of Changes to the Baswana et al. Algorithm. We now describe the changes that we make the original algorithm of Baswana et al. [7]. Full pseudocode for our algorithm is given in Algorithms 1 and 2.

- To simplify the algorithm, we remove the lists O_v, $\mathcal{E}^i_v$, and the notation of *ownership*. (Note that the original algorithm also could be changed in this way.) Instead, we keep for each vertex v the following sets in a dynamic hash table and also maintain their respective sizes: (a) For each level $j > \text{LEVEL}(v)$ the set $N_{=j}(v)$, i.e., all edges incident to neighbors on level j, and (b) one set containing the set $N_{<\text{LEVEL}(v)+1}(v)$, i.e., all edges incident to neighbors on level $\text{LEVEL}(v)$ and below.
- Invariants 2–4 are exactly the same as above, Invariant 1 is no longer needed as we no longer use the concept of ownership.
- Define C to be a sufficiently large constant used by the algorithm.
- Whenever the algorithm executes CheckForRise(v, i) for a vertex v with $\text{LEVEL}(v) < i$, the algorithm: **(1)** performs 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 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 PROCESS-FREE-VERTICES of Reference [7], as well as the paragraph "Handling insertion of an edge" in Section 4.2 in Reference [7].)
- **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 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 FixFreeVertex(v). Let $i = \text{LEVEL}(v)$: If $|N_{<i}(v)| \geq 4^i/(32C \log(n))$, then the algorithm executes RandomSettle(v, i), and if $|N_{<i}(v)| < 4^i/(32C \log(n))$, then it executes Fall$(v, i)$. (Our version modifies line 5 of Procedure FALLING of Reference [7].)

- We will keep a Boolean RESPONSIBLE(v) for each vertex v, which is set to True if the matching edge (v, w) was chosen during RANDOMSETTLE(v, ℓ). In this case, we say that v is *responsible* for the matched edge. If v is free, then RESPONSIBLE(v) is False. Each matching edge will have exactly one endpoint with RESPONSIBLE(v) set to True, and free vertices will always be set to False.
- We make the following change for processing an adversarial insertion of edge (u, v): The algorithm executes RESETMATCHING(u) with probability $p^{\text{reset}}_{\text{LEVEL}(u)}$ and it executes RESETMATCHING (v) with probability $p^{\text{reset}}_{\text{LEVEL}(v)}$, where $p^{\text{reset}}_i = 1/4^{i+3}$. If RESPONSIBLE($u$) = True, then RESETMATCHING(u) simply picks a new matching edge for u by removing edge $(u, \text{MATE}(u))$ from the matching and then calling FIXFREEVERTEX(u) and FIXFREEVERTEX(MATE(u)); if RESPONSIBLE(u) is False, then RESETMATCHING(u) does nothing. (Our version modifies the paragraph "Handling insertion of an edge" in Section 5.2 of Reference [7].)
- We also make the following change to procedure FALL(v, i): All edges of the set $N_{<i}(v)$ are traversed, not just the ones owned by v. Since $|N_{<i+1}(v)| = O(4^{i+1})$ (by Invariant 3), the running time analysis of Reference [7] remains valid. Furthermore, recall that as a result of v falling to level $i-1$, v now belongs to $N_{=i-1}(u)$ for every neighbor u of v at level $i-1$. Each such neighbor u then executes RESETMATCHING(u) with probability $p^{\text{reset}}_{\text{LEVEL}(u)}$. (Our version modifies lines 3 and 4 in Procedure FALLING of Reference [7].)

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., RISE, FALL, FIXFREEVERTEX, RESETMATCHING) call each other. We note that in the pseudocode, whenever the algorithm changes the level of a vertex in the hierarchy it also performs straightforward *bookkeeping work* that adjusts all sets $N_{<i}(v)$ and $N_{=i}(v)$ to match the new hierarchy. For example, if a vertex falls from level i to level $i-1$, then for every edge (v, w) with $w \in N_{<i+1}(v)$, we do the following: if LEVEL(w) = i, then we transfer w from $N_{<i+1}(v)$ to $N_{=i}(v)$; if LEVEL(w) = $i-1$, then we transfer w from $N_{=i}(w)$ to $N_{<i}(w)$ and from $N_{<i+1}(v)$ to $N_{<i}(v)$; and if LEVEL(w) < $i-1$, then we transfer w from $N_{=i}(w)$ to $N_{=i-1}(w)$ and from $N_{<i+1}(v)$ to $N_{<i}(v)$. Note that by Invariant 3 $\phi_{i+1}(v) < 4^{i+1} = O(4^{\text{LEVEL}(v)})$ and, thus, the bookkeeping can 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 FIXFREEVERTEX(v) for vertices v in our algorithm are not executed immediately. Instead, we maintain a global FIFO queue Q of vertices v for which we still need to perform FIXFREEVERTEX(v), implemented as a doubly-linked list. To avoid adding the same vertex to the queue twice and enable us to a vertex in the queue at the end of the queue, we store at every vertex a pointer to its position in the queue. If a vertex is not in the queue, then this pointer is set to NIL.

4.3 Correctness of the Modified Algorithm

To show the correctness of the modified algorithm, we need to show that it fulfills Invariants 2–4 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 2–4, and Matching Property* hold before and after the processing of each edge update. Also, for every free vertex v, we have* RESPONSIBLE(v) *is False, and for any matching edge* (v, w), RESPONSIBLE(v) *is True if and only if* (v, w) *was last chosen to enter the matching during a call to* RANDOMSETTLE($v, \cdot$)*; as a consequence, exactly one of* RESPONSIBLE(v) *and* RESPONSIBLE(w) *is True.*

ALGORITHM 1: Fully Dynamic Maximal Matching Algorithm

```
1  p_i^rise ← C log(n)/4^i for some large constant C
2  p_i^reset ← 1/4^{i+3} Initialize empty queue Q
3  Procedure Delete(u, v)                          // Process deletion of edge (u, v)
4      Perform bookkeeping work for deletion of (u, v)
5      if (u, v) ∈ M then
6          Perform bookkeeping work for deletion of (u, v)
7          Set responsible(u) and responsible(v) to False
8          Add u to end Q (or move u to end if it is already in Q)
9          Add v to end of Q (or move v to end if it is already in Q)
10         ProcessQueue()
11 Procedure Insert(u, v)                          // Process insertion of edge (u, v)
12     Perform bookkeeping work for insertion of (u, v)
13     foreach j > max{level(u), level(v)} in increasing order do
14         CheckForRise(v, j)
15         CheckForRise(u, j)
16     With probability p^reset_{level(u)} do ResetMatching(u)
17     With probability p^reset_{level(v)} do ResetMatching(v)
18     ProcessQueue()
19 Procedure ProcessQueue()
20     while Q is not empty do
21         Pop the first vertex v in Q
22         FixFreeVertex(v)
```

Proof. *Responsibilities:* The claims about responsible(v) follow trivially from the pseudocode, as we always explicitly maintain these properties.

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 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 RandomSettle 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, then v fulfills Invariant 2. If this is still the case, then the main body of FixFreeVertex(v) is executed, which either matches v with RandomSettle(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 RandomSettle 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

ALGORITHM 2: Fully Dynamic Maximal Matching Algorithm

Procedure ResetMatching(v) // Called only if level(v) > -1
  if responsible(v) = *False* **then**
    Exit Procedure ResetMatching
  $w \leftarrow$ mate(v)
  $\mathcal{M} \leftarrow \mathcal{M} \setminus \{(v, w)\}$ // Unmatch v and w
  Set responsible(v) to False // Note: responsible(w) was already False, since v was responsible for (v, w)
  Add v to end of Q (or move v to end if it is already in Q)
  Add w to end of Q (or move w to end if it is already in Q)
Procedure FixFreeVertex(v)
  $i \leftarrow$ level(v)
  Compute $N_{<i}(v)$ from $N_{<i+1}(v)$
  if $i > -1$ **and** v *is unmatched* **then**
    if $|N_{<i}(v)| \geq 4^i/(32C \log(n))$ **then**
      Compute $N_{<i}(v)$
      RandomSettle(v, i)
    else
      Fall(v, i)
Procedure RandomSettle(v, i) // Called only if $|N_{<i}(v)| \geq 4^i/(32C \log(n))$
  Pick $w \in N_{<i}(v)$ uniformly at random
  Rise (w, level(w), i)
  $\mathcal{M} \leftarrow \mathcal{M} \cup \{(v, w)\}$ // Match v and w
  responsible(v) $\leftarrow$ True
Procedure Fall(v, i)
  Compute $N_{<i}(v)$ and $N_{=i}(v)$ from $N_{<i+1}(v)$
  Perform bookkeeping work to move v from level i to level $i-1$
  foreach $w \in N_{<i}(v)$ **do**
    CheckForRise(w, i) // v joins $N_{<i}(w)$
  foreach $w \in N_{=i-1}(v)$ **do**
    With probability p_{i-1}^{reset} **do** ResetMatching(w)
  Add v to end of Q (or move v to end if v is already in Q)
Procedure CheckForRise(v, i) // Called when $|N_{<i}(v)|$ increases for $i >$ level(v)
  if $|N_{<i}(v)| \geq 4^i$ **then** Rise(v, level(v), i) // Threshold rise
  else With probability p_i^{rise} **do** Rise(v, level(v), i) // Probabilistic rise
Procedure Rise(v, i, j)
  if $\exists (v, w) \in \mathcal{M}$ **then** // Check if v is matched with some neighbor w
    $\mathcal{M} \leftarrow \mathcal{M} \setminus \{(v, w)\}$ // Unmatch v and w
    Set responsible(v) and responsible(w) to False
    Add w to end of Q (or move w to end if it is already in Q)
  Perform bookkeeping work to move v from level i to level j
  Add v to end of Q (or move v to end if it is already in Q)

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 it also holds after the current edge update. Let v be a vertex. The set $N_{<j}(v)$ increases only if (i) a neighbor w drops from a level at or above j to a level below j or (ii) an edge incident to v is inserted. We show next that Invariant 3 holds in either case. In case (i), 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 violate the invariant, namely, $N_{<j}(v)$ and in this case $|N_{<j}(v)| = 4^j$. The fall of w from level j calls CHECKFORRISE(v, j), which in turn immediately calls RISE$(v,\text{LEVEL}(v),j)$ 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. In case (ii) in the insertion operation the function CHECKFORRISE(v, j) is called for every level j that is larger than the level of v. If for one of these levels, let us call it i, $|N_{<i}(v)| \geq 4^i$, then v is moved up to the highest level j for which $|N_{<j}(v)| \geq 4^j$. Thus, Invariant 3 is guaranteed.

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 RANDOMSETTLE 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.

Matching Property:* Finally Matching Property* 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 RANDOMSETTLE(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 Matching Property* holds. □

Remark 4.3. We later prove a stronger version of Lemma 4.2, which shows that Invariant 4 and relaxed version of Invariants 3 hold not just at the end of processing an adversarial update, but also at all points in the middle of processing an update. See Lemma 4.11 for more details.

4.4 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, then the algorithm must do bookkeeping work to maintain the various sets $N_{<\text{LEVEL}(v)+1}(v)$ and $N_{=j}(v)$ 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, RANDOMSETTLE, 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 $N_{<i+1}(v)$, which by Invariant 3 contains at most $O(4^{i+1})$ edges. We now analyze the bookkeeping required for procedure Rise:

Lemma 4.4. Rise(v, i, j) *requires* $O(4^j)$ *bookkeeping work.*

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. First, every neighbor u of v whose level is less than j must update $N_{=j}(v)$ and either $N_{=i}(v)$, if $\text{level}(u) < i$, or $N_{<\text{level}(u)+1}(u)$ if $i \leq \text{level}(u) < j$. By Invariant 3 there are at most $O(4^j)$ neighbors to update. Second, 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

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

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 have 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

$$\sum_{k=i}^{j-1} O(4^k) = O(4^j). \qquad \square$$

Before analyzing the recursive work, we bound the probability that CheckForRise(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. For this, we need to introduce the notion of a hierarchy. When we refer to the *(graph) hierarchy* $\mathcal{H}(t)$ at time t, we mean the current graph G, the level assigned to each vertex at time t, as well as the set of edges in the matching at time t.

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 CheckForRise, the second call to CheckForRise, the third one, and so on.

Lemma 4.5. *For any* k*, it holds with high probability that the* k*th call to* CheckForRise *does* not *lead to a threshold rise.*

Proof. Let $B_{v,i}$ be the bad event that the kth call to CheckForRise 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 kth call to CheckForRise 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, i.e., t is either the start of the algorithm or a point in time when v falls below level i. It is not hard to see that because Matching Property* only allows a vertex to fall to below level i when $|N_{<i}(v)| < 4^i/(32C\log(n))$, it must be the case that at time t, we have $\phi_v(i) < 4^i/(32C\log(n)) < 4^i/2$. (There is also the fringe case $t = 0$; in this case $\phi_v(i) = 0$ at time t, because in the fully dynamic setting one can assume w.l.o.g. that the graph starts empty.) Thus, there must have been at least $4^i/2$ calls to CheckForRise(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 CheckForRise(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(v), it is *indirectly* called by the procedure that added v to the queue; we say that the called procedure is *caused* by the calling procedure and, in case of FixFreeVertex(v), we say that it is *caused* by the last procedure that affected v's position on the queue, either by placing it on the queue or by moving it to the end. For example, a procedure Fall leads to many calls to CheckForRise, and so can potentially lead to many calls to procedure Rise. We can thus construct a *causation tree* for each adversarial edge update, whose root is the procedure handling the adversarial edge update and 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 (recursively) 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 any call to Fall(v, i), RandomSettle(v, i), or FixFreeVertex(v) is at most $O(4^i)$. The running time of the remaining procedures ResetMatching, CheckForRise, and Rise can then be easily analyzed in terms of the analysis of the earlier three procedures. Note that the time to process any procedure at time t depends on two things: the hierarchy at time t and the random coin flips made after time t. Thus, we can define $\mathbf{E}_i^{\text{fall}}(v, \mathcal{H}(t))$ to be the expected total work to process Fall(v, i) given that the state of the current hierarchy is $\mathcal{H}(t)$, where the expectation is taken over all coin flips made after time t. (We assume that in the hierarchy $\mathcal{H}(t)$ vertex v has level i, since otherwise Fall(v, i) is not a valid procedure call.) We define $\mathbf{E}_i^{\text{settle}}(v, \mathcal{H}(t))$ and $\mathbf{E}_i^{\text{free}}(v, \mathcal{H}(t))$ analogously. We say that some hierarchy $\mathcal{H}(t)$ 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 $\mathbf{E}_i^{\text{fall}}$ be the maximum of all $\mathbf{E}_j^{\text{fall}}(v, \mathcal{H})$, where the maximum is taken over all levels $j \le i$, all vertices v, and all valid hierarchies $\mathcal{H}$ in which v has level j. Define $\mathbf{E}_i^{\text{settle}}$ and $\mathbf{E}_i^{\text{free}}$ accordingly. Define $\mathbf{E}_i^{\max} = \max\{\mathbf{E}_i^{\text{fall}}, \mathbf{E}_i^{\text{settle}}, \mathbf{E}_i^{\text{free}}\}$. Note that because $\mathbf{E}_i^{\max}$ 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 $\mathbf{E}_i^{\max}$.

Lemma 4.6. $\mathbf{E}_i^{\max} \le O(4^i) + 3\mathbf{E}_{i-1}^{\max}$

Proof. We first show that $\mathbf{E}_i^{\text{settle}} \le O(4^i) + \mathbf{E}_{i-1}^{\max}$. RandomSettle($v, i$) picks some random mate v' for v with $\text{level}(v') < i$, performs $O(4^i)$ bookkeeping work to move v' to level i (Lemma 4.4), 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 $\mathbf{E}_{i-1}^{\max}$.

Now consider FixFreeVertex(v), where $i = \text{level}(v)$. The work of this procedure is to construct $N_{<i+1}(v)$ from $N_{<i+1}(v)$ is $O(4^i)$. The algorithm then causes one other procedure call: either RandomSettle(v, i) or Fall(v, i), depending on the size of $N_{<i}(v)$. We have already bounded $\mathbf{E}_i^{\text{settle}}$, so all that remains is to bound $\mathbf{E}_i^{\text{fall}}$.

Recall that the algorithm only executes Fall(v, i) when $N_{<i}(v) < 4^i/(32C\log(n))$. The procedure Fall requires the standard $O(N_{<i+1}(v)) = O(4^i)$ bookkeeping work, and it also causes a call to FixFreeVertex(v) at level $i-1$, which has $\mathbf{E}_{i-1}^{\max}$ expected total work. Fall(v, i) can also lead to additional updates at level $i-1$ due to ResetMatching: See Line 51 of Algorithm 2. Finally, unlike the other procedures, Fall(v, i) can also cause additional procedure calls at level i. This can happen because each neighbor u of v at level $\le i-1$ executes CheckForRise(u, i) (line 49) (note that it is not possible that a neighbor calls a CheckForRise(u, j) for $j > i$) . This has a small chance of resulting in Rise(u, level(u), i) (either through a probabilistic rise or through a threshold rise—see Lemma 4.5), followed by FixFreeVertex(u), where $\text{level}(u) = i$, and FixFreeVertex(old-mate(u)), where $\text{level}(\text{old-mate}(u)) \le i-1$.

Let X^{reset} be the random variable that stands for the number of ResetMatching(u) triggered by the fall of v, and note that every such vertex is at level $i-1$. Let X^{rise} be the number of Rise(u, level(u), i) triggered by the fall. Note that $\mathbf{E}[X^{\text{reset}}] \leq 1/4$, because v is at level i before the fall, so by Invariant 3, v has at most 4^{i+1} neighbors at level $i-1$, and each neighbor has a $p_{i-1}^{\text{reset}} = 1/4^{i-1+3}$ probability of being reset. We now argue that $\mathbf{E}[X^{\text{rise}}] \leq 1/16$. By Matching Property*, v has at most $4^i/(32C\log(n))$ neighbors u at lower level before the fall, each of which executes CheckForRise(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 probabilistic-rise, and by Lemma 4.5 the probability of a threshold-rise is negligible (Lemma 4.5), so for simplicity, we upper bound it by $C\log(n)/4^i$. Thus: $\mathbf{E}[X^{\text{rise}}] = [4^i/(16C\log(n))][2C\log(n)/4^i] = 1/16$.

We now consider the total work to process a fall. First, 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 call to ResetMatching or Rise. Each reset causes two additional procedure calls at level $i-1$, whose running time we upper-bound by $2\mathbf{E}_i^{\max}$ (using the fact that $\mathbf{E}_{i-1}^{\max} \leq \mathbf{E}_i^{\max}$). Each Rise procedure requires $O(4^i)$ bookkeeping work and causes a new call at level i, as well as a call at another level less than i (due to the old mate w becoming free). We upper-bound the time to execute these two calls at level i or less again by $2\mathbf{E}_i^{\max}$. Note that this upper bound allows to achieve a crucial probabilistic independence: Although the value of X^{rise} might be correlated with the time to process these calls to Rise (both depend on the current hierarchy), the value of X^{rise} is *completely independent* from $\mathbf{E}_i^{\max}$, since the latter takes the maximum over all valid hierarchies and so does not depend on the current hierarchy. Now, recall that $\mathbf{E}[X^{\text{reset}}] \leq 1/4$ and $\mathbf{E}[X^{\text{rise}}] \leq 1/16$. Putting it all together, we can write a recursive formula for $\mathbf{E}_i^{\max}$.

$$\begin{aligned}
\mathbf{E}_i^{\max} &\leq O(4^i) + \mathbf{E}_{i-1}^{\max} + (2\mathbf{E}_i^{\max} + O(4^i)) \sum_{k=1}^{\infty} k\mathbf{Pr}[X^{\text{reset}} + X^{\text{rise}} = k] \\
&\leq O(4^i) + \mathbf{E}_{i-1}^{\max} + (2\mathbf{E}_i^{\max} + O(4^i))(\mathbf{E}[X^{\text{rise}} + X^{\text{reset}}]) \\
&= O(4^i) + \mathbf{E}_{i-1}^{\max} + (2\mathbf{E}_i^{\max} + O(4^i))\frac{5}{16} < O(4^i) + \mathbf{E}_{i-1}^{\max} + \frac{5}{8}\mathbf{E}_i^{\max}. \qquad \square
\end{aligned}$$

Bringing $\frac{5}{8}\mathbf{E}_i^{\max}$ to the left side of the inequality and multiplying it by 8/3 it leads to the statement of the lemma.

Corollary 4.7. *The expected total work for a call to* FixFreeVertex, Fall, RandomSettle, Rise, *or* ResetMatching *or* CheckForRise *at level i is $O(4^i)$, where* Rise(v, i', i) *is said to be a procedure call at level i.*

Proof. Solving the recurrence relation in Lemma 4.6 yields $\mathbf{E}_i^{\max} = O(4^i)$, which gives us the desired bound for FixFreeVertex, Fall, and RandomSettle. Procedure ResetMatching causes two calls to FixFreeVertex, so the same $O(4^i)$ bound applies. Procedure Rise requires $O(4^i)$ bookkeeping work (Lemma 4.4), and then causes at most two other calls to FixFreeVertex, each of which we know has expected total work $O(4^i)$. Procedure CheckForRise does $O(4^i)$ bookkeeping work and then causes at most one call to Procedure Rise at level i. Thus, the same $O(4^i)$ bound applies. $\square$

4.5 Bounding the Probability that an Edge Appears in the Matching

Now that we have analyzed the time to process the individual procedure calls, we turn our attention to the time 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 $(v, \text{MATE}(v))$ with v and $\text{MATE}(v)$ 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 (x, y) is a matching edge. Note that for (x, y) to be a matching edge, it must have been chosen by a RandomSettle(x, i) or RandomSettle(y, i) for some level i. There are thus $2(\text{L}_0 + 1) = O(2\log(n))$ possible procedure calls that could have created this matching edge: we bound the probability of each separately.

Lemma 4.8. *Let (x, y) be any edge at any time t^* during the update sequence, and let $0 \leq \ell \leq \lfloor \log_4(n) \rfloor$ be any level in the hierarchy. Then:* $\mathbf{Pr}[$*at time t^*, (x, y) is a matching edge at level ℓ and* responsible(x) *is True*$] = O(\log^3(n)/4^\ell)$*, where the probability is over all random choices made by the algorithm. (Note that this is equivalent to the probability that (x, y) was chosen by* RandomSettle(x, ℓ)*.)*

Corollary 4.9. *Let (x, y) be any edge at any time t^* during the update sequence, and let $0 \leq \ell \leq \lfloor \log_4(n) \rfloor$ be any level in the hierarchy. Then:* $\mathbf{Pr}[$*at time t^*, (x, y) is a matching edge at level ℓ*$] = O(\log^3(n)/4^\ell)$*, where the probability is over all random choices made by the algorithm.*

Proof of Corollary 4.9. For any edge (x, y) at level ℓ that is in the matching, we have that either responsible(x) or responsible(y) is True. We can thus apply Lemma 4.8 to each of those two cases and union bound the two resulting probabilities. □

The proof of this lemma is very involved, and the rest of this subsection is devoted to proving it. Let us first briefly discuss the naive approach and why it fails to work. Let t be the *last* time before t^* that RandomSettle(x,ℓ) is called, and note that assuming responsible(x) is True at time t^* the matching edge is precisely the matching edge picked at time t. Matching Property* guarantees that at any given call to RandomSettle(x,ℓ) only has a $O(\log(n)/4^\ell)$ probability of picking the specific edge (x, y). Thus, it is tempting to (falsely) argue that at time t the probability that RandomSettle(x,ℓ) picked edge (x, y) is at most $O(\log(n)/4^\ell)$. But this might not be true, because although RandomSettle(x,ℓ) picks an edge uniformly at random from many options, the fact that we condition on t being the *last* random settle before t^* means that we condition on events *after* time t, which can greatly skew the distribution at time t. Consider, for illustration, the update sequence in the star graph at the beginning of Section 4.2: The sequence repeatedly inserts and deletes all edges other than (v, v'), so any edge other than (v, 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 RandomSettle(x,ℓ) is called in some critical time period.

One of the main difficulties of the proof is that we have to be very careful with the assumption of obliviousness. The model assumes that adversarial updates are oblivious to our hierarchy, so the specific mate that v chooses in some RandomSettle(v, k) will not affect future adversarial updates. But the internal changes made by the algorithm might not be oblivious: If v chooses the specific edge (v, w), then this will change the level of w, which will lead to changes to the neighbors of w, which might indirectly increase the probability of some ResetMatching(x), which will lead to the removal of (x, y) from the matching. Thus, internal updates are adaptive to internal random choices.

To overcome this adaptivity issue, we will show that the higher levels of the hierarchy are in fact oblivious to random choices made by lower levels of the hierarchy.

Comparison to the analysis of Baswana et al. Our proof of Lemma 4.8 consists of two main parts. The first part, which includes Sections 4.5.1 and 4.5.2, establishes that higher levels of the hierarchy are independent from random choices made on lower levels. This part is very similar to an

analogous proof of independence in Baswana et al. [7] (see Lemma 4.13, Theorem 4.2, Lemma 4.17), although we use different notation that is more amenable to the second part of the proof.

In the second part of the proof, which includes Sections 4.5.3 and 4.5.4, we show that this independence allows us to prove Lemma 4.8. This part is entirely new to our article, because the claim does *not* hold for the original algorithm of Baswana et al. [7]. We show how our modifications of Reference [7], and especially our introduction of the ResetMatching procedure, allows us to replace the fundamentally amortized guarantees of Reference [7] with the universal upper bound on the probability in Lemma 4.8.

4.5.1 Hierarchy Changes and Invariants during the Processing. So far, we have only concerned ourselves with the state of the hierarchy after it completes processing some adversarial update. But the algorithm can make many changes to the hierarchy while processing only a single adversarial update. In this section, we will need notation to analyze the hierarchy in the middle of processing.

Definition 4.10. We refer to a *change* in the hierarchy as any of the following operations. All line numbers relate to Algorithm 1 or 2.

(1) Removing an edge (x, y) from the matching in Line 27 or 58;
(2) Adding an edge (x, y) to the matching in Line 43;
(3) Moving a vertex from some level i to some level $j > i$, and the associated bookkeeping changes in Line 61 in Rise $(\cdot, i, j)$;
(4) Moving a vertex from some level i to level $i - 1$, and the associated bookkeeping changes in Line 47 in Fall $(\cdot, i)$;
(5) Performing an adversarial insertion of edge (x, y) in Line 12;
(6) Performing an adversarial deletion of edge (x, y) in Line 6.

Given any execution of our dynamic algorithm let $\sigma_1, \sigma_2, \ldots$ be the sequence of all hierarchy-changes made by the algorithm.

In Lemma 4.2, we showed that right after the algorithm has completed the processing of some adversarial insertion/deletion, the hierarchy satisfies Invariants 2–4. Note, however, that if we look at the hierarchy right after some change σ_i, then Invariant 2 might be violated, since Algorithm 1 might not yet have terminated, so there might still be free vertices that need fixing. Also Invariant 3 needs to be (slightly) relaxed to Invariant 3':

- **Invariant 3':** For each vertex v and for all $j > \text{level}(v)$, $\phi_v(j) \leq 4^j$ holds true.

Thus, we now show that Invariants 3' and 4 are true in *every* instantiation of the hierarchy, i.e., at any point in the algorithm.

Lemma 4.11 (Generalized Invariants). *Let $\mathcal{H}$ be the hierarchy right after some change σ_i. Then, $\mathcal{H}$ satisfies Invariants 3' and 4 above. (Note that Matching Property* certainly continues to hold, because it corresponds to the behavior of the algorithm itself, not to any hierarchy invariant.)*

Proof. For Invariant 4, the only time we insert an edge (v, w) into the matching is in Line 43 of RandomSettle and Line 42 ensures that when we do so, we have $\text{level}(v) = \text{level}(w)$. Whenever a vertex falls in level it is a free vertex, so Invariant 4 is trivially preserved. Finally, a vertex only rises in level inside the Rise operation, and Line 58 ensures that we only perform the actual rise on a free vertex.

The proof for Invariant 3' is much more involved, though it is conceptually straightforward. Recall that Invariant 3' states that $\phi_v(i) \leq 4^i$ for all vertices v and level i; see the beginning of Section 4.2 for the definition of $\phi_v(i)$.

Define a hierarchy change to be *risky* if it increases $\phi_v(i)$ for some pair v, i. Since $\phi_v(i)$ can only increase when $|N_{<i}(v)|$ increases or when v drops from level i to $i-1$, it is easy to check that our algorithm only performs two types of risky hierarchy changes: (A) an adversarial insertion of edge (u, v) and (B) the falling of a vertex from some level j to level $j-1$. These changes to the hierarchy are made in Line 12 of Algorithm 1 and Line 47 of Algorithm 2, respectively.

Given a vertex v and a level i, we say that pair v, i is *violating* if $\phi_v(i) \geq 4^i$. Note that only a risky operation can cause a pair to become violating. We now prove Invariant 3 via induction on the number of risky hierarchy changes.

Induction Hypothesis: Consider any risky hierarchy change σ. Then, right before σ, all pairs v, i are non-violating; that is, we have $\phi_v(i) < 4^i$ for every vertex v and level i.

Induction Basis: The proof of the base case is trivial: When the algorithm begins, we have $\phi_v(i) = 0$ for every pair v, i, and this continues to hold before the first risky change, because non-risky changes cannot increase any $\phi_v(i)$.

Induction Step: Assume that the induction hypothesis holds for some risky change σ. We now show that it also holds for the next risky change σ'. We consider two cases:

Case 1: σ corresponds to Line 47 in Procedure Fall **(v, i).** Note that the algorithm performs no risky hierarchy changes between executing Lines 47 and 52 (including all the sub-routines called in between). Thus, it is sufficient to show that by the time the algorithms finish Procedure Fall (v, i) in Line 52 there are no violating pairs; this will then continue to hold until the next risky operation σ', because the non-risky operations in between cannot create new violating pairs or cause already fixed pair to become violating again. Note that because all the hierarchy changes made during Fall (v, i) (e.g., the rising of vertices) are non-risky, it is in fact enough to show that every pair w, j is or becomes non-violating at some point between Lines 47 and 52.

The effect of the hierarchy change σ that causes v to fall from level i to level $i-1$ is to increase $\phi_w(i) = |N_{<i}(w)|$ by 1 for every neighbor w that is in $N_{<i}(v)$. By the induction hypothesis, we then have $\phi_w(i) \leq 4^i$ for all such pairs, while all other pairs remain non-violating. Now, for every $w \in N_{<i}(v)$ the Fall (v, i) operation executes CheckForRise (w, i) (Line 49). When the algorithm executes CheckForRise (w, i), if $\phi_w(i) < 4^i$, then we are done, because as argued above it is enough to show that w, i is non-violating at some point during the execution of Fall (v, i). If $\phi_w(i) = 4^i$, then the algorithm raises w to level i (threshold rise in CheckForRise (w, i)), so after the rise, we have level$(w) = i$ and $\phi_w(i) = 0 < 4^i$, as desired.

Case 2: σ corresponds to Line 12 Procedure Insert **(u, v).** Note that the algorithm performs no risky hierarchy changes between Line 12 and Line 16 (including subroutines called by lines in between). Thus, analogously to the previous case, it is enough to show that every pair w, j is or becomes non-violating at some point between the execution of Line 12 and Line 16. The effect of the hierarchy change σ that inserts edge (u, v) is to increase $\phi_u(j) = |N_{<j}(u)|$ and $\phi_v(j) = |N_{<j}(v)|$ for every $j > \max\{\text{level}(u), \text{level}(v)\}$. We focus on the pairs u, j, since the pairs v, j are analogous. For every such pair u, j the algorithm executes CheckForRise (u, j) (Line 14). As in Case 1, if $\phi_u(j) < 4^j$ when the algorithm executes CheckForRise (u, j), then pair u, j is non-violating and we are done. If $\phi_u(j) = 4^j$, then again as in Case 1, the algorithm raises u to level j, which leads to $\phi_u(j) = 0 < 4^j$, so j, u becomes non-violating. □

4.5.2 Upper and Lower Hierarchy and Hierarchical Independence. We now formally separate changes to the upper and lower parts of the hierarchy and then show the independence of the upper hierarchy from the lower hierarchy.

Definition 4.12. We say that a hierarchy change σ is *above* level ℓ if one of the following holds:

(1) The change removes/adds an edge (u, v) from/to the matching for which level$(u) > \ell$. (Recall that level(u) = level(v) by Invariant 4.)

(2) The change raises a vertex from level i to level $j > i$ with $j > \ell$ (it does not matter if $i > \ell$).
(3) The change moves a vertex from level i to level $i - 1$ with $i > \ell$.
(4) The change is an adversarial insertion/deletion of edge (x, y) (regardless of $\text{LEVEL}(x)$ and $\text{LEVEL}(y)$).

For any execution of the dynamic algorithm, let $S = \sigma_1, \sigma_2, \ldots$ be the sequence of changes made to the hierarchy. Let $S^\ell = \sigma_1^\ell, \sigma_2^\ell, \ldots$ be the subsequence of S consisting of all changes at level $> \ell$: That is, S^ℓ contains all σ_i above level ℓ, in the same order as in S.

To prove independence of the hierarchy above level ℓ, we will think of the algorithm as using two different random bit-streams.

Defining higher and lower random bits: Let $\mathcal{A}$ be the sequence of updates made by the adversary: Since the adversary is oblivious, we can fix this sequence in advance. Let B be the entire sequence of random bits used by the algorithm. If the algorithm is given all of $(\mathcal{A}, B)$ as input, then it will always produce the same hierarchy.

Now, let us conceptualize the same algorithm a bit differently. Let ℓ be the fixed level in the statement of Lemma 4.8. We will have two sequences of random bits: $B_{>\ell}$ and $B_{\leq\ell}$. Whenever the algorithm runs $\text{RANDOMSETTLE}(v, k)$ for any $v \in V$, if $k > \ell$, then it chooses the new random mate for v using the bits in $B_{>\ell}$; otherwise, it uses the bits in $B_{\leq\ell}$. Similarly, whenever the algorithm executes $\text{RISE}(v, \text{LEVEL}(v), k)$ with probability p_k^{rise}, the random bits for p_k^{rise} are taken from $B_{>\ell}$ if $k > \ell$, and from $B_{\leq\ell}$ otherwise. Finally, whenever the algorithm executes $\text{RESETMATCHING}(v)$ with probability $p^{\text{reset}}_{\text{LEVEL}(v)}$, the random bits for $p^{\text{reset}}_{\text{LEVEL}(v)}$ are taken from $B_{>\ell}$ if $\text{LEVEL}(v) > \ell$ and from $B_{\leq\ell}$ otherwise.

Note that the execution of the algorithm is completely determined by the triplet $(\mathcal{A}, B_{>\ell}, B_{\leq\ell})$. Previously, we only fixed the update sequence $\mathcal{A}$ and assumed the bits in $B_{>\ell}, B_{\leq\ell}$ were chosen randomly. We now modify this as follows: Given any fixed sequence of bits B^+, we say that *the algorithm run with $B_{>\ell}$ set to B^+* if the bits from $B_{>\ell}$ are always taken from B^+, but the bits from $B_{\leq\ell}$ are chosen randomly. Then, when we speak of probabilities in such an execution, the probability is only over the bits in $B_{\leq\ell}$.

The lemma and its corollary below formally states the independence of the hierarchy above level ℓ. Intuitively, the lemma says the following: Fix a sequence of adversarial updates $\mathcal{A}$ and a sequence of bits B^+, and say that we run our dynamic matching algorithm with $B_{>\ell}$ set to B^+. Then, the sequence S^ℓ of changes above ℓ is *deterministically* determined by B^+; in other words, the sequence will always be the same regardless of the random bits in $B_{\leq\ell}$. There is one caveat: When a vertex v rises from level i to $j > \ell$ (change type 2 in Definition 4.12), although v and j are always deterministically determined by B^+, i may depend on $B_{\leq\ell}$ if $i \leq \ell$.

LEMMA 4.13 (HIERARCHICAL INDEPENDENCE). *Let B_1, B_2 be any two bit sequences for $B_{\leq\ell}$. Let $\sigma_1^\ell, \ldots$ be the sequence of changes above ℓ performed by the execution that uses $\mathcal{A}, B^+, B_1$ and let $\tau_1^\ell, \ldots$ be the sequence of changes above ℓ performed by the execution that uses $\mathcal{A}, B^+, B_2$. Then, every σ_k^ℓ is equivalent to τ_k^ℓ in the following sense:*

- *If σ_k^ℓ moves vertex v from level i to level $j > i$ (with $j > \ell$), then τ_k^ℓ moves the same vertex v from level i' to the same level $j > i'$; moreover, $i \neq i'$ is only possible if $i \leq \ell$ and $i' \leq \ell$.*
- *If σ_k^ℓ is any other type of change above ℓ, then σ_k^ℓ is identical to τ_k^ℓ.*

PROOF. Although it is somewhat involved, conceptually speaking, the proof is quite simple: We go through the possible operations of the algorithm and show by induction that because both algorithms use the same bits B^+ for $B_{>\ell}$, the two executions always have the same above-ℓ hierarchy.

Intuitively, the induction hypothesis is that all times, both executions have the same above-ℓ hierarchy as well as the same queue Q of free vertices above level ℓ (see Algorithm 1). The issue is that when we say "at all times," this does not refer to execution time, since one algorithm may be ahead of the other. Instead, as in the lemma statement, we formalize the intuition by talking about the sequence of above-ℓ changes and of changes to Q.

Defining Relevant Operations: Let A_i be the execution of the algorithm running on $\mathcal{A}, B^+, B_i$ for $i = 1, 2$. Consider the following sequence of operations $\gamma_1, \gamma_2, \ldots$ consisting of (i) above-ℓ changes and (ii) changes (addition/removal/move-to-end) to the queue formed by this execution. Whenever the execution makes an above-ℓ hierarchy-change, we add this change to the end of the sequence. Similarly, whenever the execution adds or removes a vertex v to queue Q or moves v to end of Q with $\textsc{level}(v) > \ell$, we add this change to the end of the sequence. We refer to the γ_i as *relevant* operations. Let A_2 be the execution running on $\mathcal{A}, B^+, B_2$ and define relevant operations $\delta_1, \delta_2, \ldots$ analogously for this execution. We say that $\gamma_i = \delta_i$ if one of the following holds:

- Both γ_i and δ_i add/remove/move-to-end the same vertex v in Q and v has the same level when γ_i is performed in execution A_1 as when δ_i is performed in execution A_2.
- γ_i and δ_i correspond to the same non-rise hierarchy change.
- Both γ_i and δ_i raise the same vertex v to the same level above ℓ.

Claim 4.14 (Main Claim). For every i, we have $\gamma_i = \delta_i$.

It is easy to verify that the main claim proves Lemma 4.13, as it implies that the above-ℓ hierarchy and the above-ℓ free vertices evolve in the same way, which is strictly stronger than the lemma statement, which only concerns the hierarchy changes. We prove the claim by induction.

Induction Hypothesis: For any i, for all $j \leq i$ it holds that $\gamma_j = \delta_j$ and that both executions A_1 and A_2 have used the same number of bits from B^+.

Induction Bases: The base case is trivially true, since the graph starts empty, so γ_1 and δ_1 both correspond to the same adversarial insertion.

Induction Step: To show the inductive step, we will consider somewhat larger chunks of the algorithm. We note that all relevant operations performed by the algorithm—whether hierarchy changes or changes to Q—occur in one of three scopes in Algorithm 1.

Definition 4.15. Define the delete-body of Algorithm 1 to consist of the execution of all lines in Procedure $\textsc{Delete}(u, v)$ *except* $\textsc{ProcessQueue}()$ in Line 10, including all subroutines called during the execution of these lines. Define the insert-body to consist of the execution of all of Procedure $\textsc{Insert}(u, v)$ except $\textsc{ProcessQueue}()$ in Line 18, including all subroutines called during the execution of these lines. Finally, define the fix-body to contain Lines 21–22, including all subroutines called during the execution of these lines. We say that the fix-body is above level ℓ if the vertex v popped from Q has $\textsc{level}(v) > \ell$.

Definition 4.16. We say that a relevant operation γ_i or δ_i is *primary* if it is an adversarial insertion, an adversarial deletion, or the removal of a vertex v from Q with $\textsc{level}(v) > \ell$ (this last change always occurs in Line 21). (Note that only a removal from Q counts as a primary change: not an insertion or a move in Q.)

Observation 4.17. *All relevant operations performed by the algorithm occur within either an insert-body, a delete-body, or a fix-body above level ℓ. Moreover, each of these three bodies always begins with a primary change γ_i, and each primary change initiates the corresponding body of the algorithm.*

By the observation above, the proof of the Main Claim consists of three possible cases summarized in the following claim:

Claim 4.18. Assume that $\gamma_k = \delta_k$ is an adversarial deletion or insertion of some edge (u, v), that the main claim holds for every $i \leq k$, and that both executions A_1 and A_2 have used the same number of bits from B^+ up to operation γ_k, respectively, δ_k. Let γ_q be the next primary change performed by execution A_1. Then $\gamma_r = \delta_r$ for each $r \in [k, q]$; that is, the main claim holds for every $i \leq q$, and that and that both executions A_1 and A_2 have use the same number of bits from B^+ between operation up to operation γ_q, respectively, δ_q.

The same holds if $\gamma_k = \delta_k$ pops some vertex v from Q with $\textsc{level}(v) > \ell$.

It is easy to check that the claim above proves the Main Claim. We first start with a definition and a couple observations.

Definition 4.19. For any j, let $\mathcal{H}_j^1$ and Q_j^1 be the above-ℓ hierarchy of execution A_1 and the (ordered) queue of above-ℓ vertices after $\gamma_1, \ldots, \gamma_j$ have been performed. Define $\mathcal{H}_j^2$ and Q_j^2 analogously.

Observation 4.20. *Say that the main claim holds for all $i \leq k$. Then, $\mathcal{H}_k^1 = \mathcal{H}_k^2$ and $Q_k^1 = Q_k^2$. Moreover, if a vertex v has $\textsc{level}(v) > \ell$ in $\mathcal{H}_k^1 = \mathcal{H}_k^2$, then the bit $\textsc{responsible}(v)$ is the same in both hierarchies.*

Observation 4.21. *Consider any fix-body that is* not *above level ℓ. Then, during the execution of these lines, no relevant changes can occur. This observation can be verified by looking at the flow of our algorithm, and observing that fixing a vertex at level $\leq \ell$ can only make hierarchy changes at level $\leq \ell$ and add/move vertices to Q with level $\leq \ell$.*

Proof of Claim 4.18. (A) We first show the claim for a deletion, as this is the simplest case. Note that by Observation 4.20, whether or not $(u, v) \in \mathcal{M}$ will be the same in both executions. Thus, either both executions add u and v to the end of Q, or they make no relevant changes, so both executions make the same relevant changes in the delete-body, and so continue to have the same above-ℓ hierarchy and queue.

Now, if the above-ℓ queue is empty at the end of this delete-body, then by Observation 4.21, the next primary change in both executions will be an adversarial insertion/deletion from $\mathcal{A}$, which is clearly the same in both executions. Else, since Q is the same in both executions after the delete-body, if v is the first vertex in Q with $\textsc{level}(v) > \ell$, then the popping of v will be the next primary operation in both executions. Either way, the next primary operation $\gamma_q = \delta_q$ is the same in both executions and neither uses any bits of B^+.

(B) We next show the claim for the case of an insertion. In Line 13 the algorithm computes $j = \max\{\textsc{level}(u), \textsc{level}(v)\}$. Note that, since the above-ℓ hierarchies are the same in both executions when the insert-body begins (Observation 4.20), if $j > \ell$, then j will be the same in both executions. However, if $j \leq \ell$, then no relevant operations will occur in Lines 13–15 and we only have to analyze the ResetMatching operations in Line 16 and Line 17 (see below). Thus, we can assume that $j > \ell$ and that both executions now execute lines $\textsc{CheckForRise}(v, j)$ and $\textsc{CheckForRise}(u, j)$. We now focus on $\textsc{CheckForRise}(v, j)$; the argument for u is identical.

The first line of $\textsc{CheckForRise}(v, j)$ checks if $N_{<j}(v) \geq 4^j$: Since the above-ℓ hierarchies of A_1 and A_2 are the same and since $j > \ell$, it is follows that $N_{<j}(v)$ is the same in both executions. Thus, either both executions will perform the same threshold rise or neither will. The next line of $\textsc{CheckForRise}(v, j)$ performs a rise with probability p_i^{rise}. Since $j > \ell$, the bits used to determine this probabilistic rise come from B^+, and so are the same in both executions. Note that both executions access the same bit from B^+, as by the induction hypothesis both executions have looked at the same bits in B^+ so far. Thus, again, either both operations perform the same rise and the corresponding relevant changes or neither do. They also consume the same number of bits from B^+.

Afterwards they will use the same bits in B^+ to decide whether to execute a ResetMatching(u) and/or ResetMatching(v). Within a ResetMatching(w) exactly the same operations will be executed as the same vertices are responsible for an edge (Observation 4.20).

Thus, the above-ℓ hierarchy and queue continue to be the same in both executions throughout the insert-body and the number of bits of B^+ that are used is identical. By the same argument as in (A), the next primary operation will also be the same.

(C) Finally, we show the claim if δ_k pops some vertex v from Q with level(v) $> \ell$. In this case, the algorithm executes FixFreeVertex(v) in both executions. Recall that we assume in the lemma statement that $i =$ level(v) $> \ell$. Thus, since the above-ℓ hierarchies are the same at the beginning of the fix-body (Observation 4.20), we know that when we compute sets $N_{<i}(v)$ and $N_{<i+1}(v)$ in Line 33 of Algorithm 2, each set will be the same in both executions. Thus, either they will both execute the If statement of Line 35 or they will both execute the Else statement of Line 39.

If they both execute the If statement, then they both execute RandomSettle(v, i). Note that the random mate w picked by this RandomSettle will be the same in both executions because $i > \ell$, so w is picked according to bits in B^+, which are the same in both executions. Thus, in the case of the If statement, it is easy to check that all relevant operations performed by the fix-body will be the same in both executions and the number of consumed bits of B^+ is identical.

Now, say that the algorithm instead executes Fall(v, i) from the Else statement. Then the algorithm performs CheckForRise(w, i) for every $w \in N_{<i}(v)$. Because $N_{<i}(v)$ are the same in both executions, the same operations CheckForRise(w, i) are performed. By the same argument as in (B), these CheckForRise(w, i) then lead to the same relevant operations $\gamma_r = \delta_r$ in both executions and use the same number of bits from B^+. The algorithm then performs ResetMatching(w) for each $w \in N_{=i-1}(v)$. Once again, if $i - 1 \leq \ell$, then none of these operations are relevant, so the claim trivially holds. Otherwise, $i - 1 > \ell$ and, since the above-ℓ hierarchy is the same in both executions up to this point, we have that $N_{=i-1}(v)$ is the same in both executions and they will both use the same bits and the same number of bits from B^+ to decide whether to perform a ResetMatching(w). Within a ResetMatching(w) exactly the same operations will be executed as the same vertices are responsible for an edge. Thus, they will have identical results in both executions.

We have thus shown that throughout the fix-body, both executions perform the same relevant operations, have the same above-ℓ hierarchy and queue, and use the same number of bits from B^+. By the same argument as in (A), the next primary operation will also be the same. □

We have thus proved the main claim and completed the proof of Lemma 4.13. □

Throughout our analysis, we will rely on the corollary below, which is an extension of Lemma 4.13. We start with an informal description.

Informal Description of Corollary 4.22: Say that we run our algorithm on a fixed update sequence $\mathcal{A}$ and that at some point during the algorithm, we call procedure RandomSettle(x, ℓ). Let $N_{<\ell}(x) = \{y_1, \ldots y_k\}$ when we call the procedure. Let Y_i be the event that RandomSettle(x, ℓ) picks y_i as the mate of x; since the mate of x is chosen uniformly at random from $N_{<\ell}(x)$, we know that $\mathbf{Pr}[Y_i] = 1/k$. This statement remains true no matter what happened before this execution of RandomSettle(x, ℓ), since RandomSettle uses fresh randomness. However, say that $\mathcal{E}$ is some event that depends on the *entire* sequence of hierarchy changes made by the algorithm, including those after the call to RandomSettle. In this case, we might have $\mathbf{Pr}[Y_i \mid \mathcal{E}] \neq \mathbf{Pr}[Y_i]$; for example, since $\mathcal{E}$ depends on the entire sequence, it might be the case that $\mathcal{E}$ can be true only if Y_i is true. The crux of our corollary is that if we consider an event $\mathcal{E}_{>\ell}$ that depends on all hierarchy changes before the call to RandomSettle(x, ℓ) and also on future above-ℓ changes made by the

algorithm, then we can indeed state that $\mathbf{Pr}[Y_i \mid \mathcal{E}_{>\ell}] = \mathbf{Pr}[Y_i] = 1/k$. The same is true of a call to ResetMatching(x), as long as level$(x) \leq \ell$ when the call is made.

Corollary 4.22. *Fix some adversarial update sequence $\mathcal{A}$, and consider the execution of the algorithm on $\mathcal{A}$. The following two properties hold:*

- *Consider some call to* RandomSettle(x, ℓ)*, let $N_{<\ell}(x) = \{y_1, \ldots y_k\}$ when the call is made, and let Y_i be the event that y_i is chosen as* mate(x) *as a result of the call. Let S_{past} be the sequence of* all *hierarchy changes before this call to* RandomSettle(x, ℓ) *and let S^{ℓ} be the sequence of all above-ℓ hierarchy changes after the call. Let $\mathcal{E}_{>\ell}$ be any event that depends only on S_{past} and S^{ℓ}; that is, whether $\mathcal{E}_{>\ell}$ is true or false is uniquely determined by these sequences. Then, for all $1 \leq i \leq k$, $\mathbf{Pr}[Y_i \mid \mathcal{E}_{>\ell}, S_{past}] = \mathbf{Pr}[Y_i \mid S_{past}] = 1/k$.*
- *Assume that at some point, we reach a line that executes* ResetMatching(u) *with some probability p (Line 16 of Algorithm 1 or Line 51 of Algorithm 2) and that at this point,* level$(u) \leq \ell$*. As above, let S_{past} be the sequence of all hierarchy changes before this line, let S^{ℓ} be the sequence of all above-ℓ hierarchy changes after the call, and let $\mathcal{E}_{>\ell}$ be any event that depends only on S_{past} and S^{ℓ}. Let Y be the event that the* ResetMatching(u) *is in fact executed. Then $\mathbf{Pr}[Y \mid \mathcal{E}_{>\ell}, S_{past}] = \mathbf{Pr}[Y \mid S_{past}] = p$.*

Proof. The proof follows easily from Lemma 4.13. We will only prove the first property about RandomSettle(x, ℓ); the property about ResetMatching(x) can be proved in the same fashion.

Since RandomSettle(x, ℓ) picks a random mate using fresh randomness that is independent from all previous choices made by the algorithm, we have $\mathbf{Pr}[Y_i \mid S_{\text{past}}] = 1/k$. By Bayes' law, we have

$$\mathbf{Pr}[Y_i \mid \mathcal{E}_{>\ell}, S_{\text{past}}] = \mathbf{Pr}[Y_i \mid S_{\text{past}}] \frac{\mathbf{Pr}[\mathcal{E}_{>\ell} \mid Y_i, S_{\text{past}}]}{\mathbf{Pr}[\mathcal{E}_{>\ell} \mid S_{\text{past}}]}.$$

We now show that $\mathbf{Pr}[\mathcal{E}_{>\ell} \mid Y_i, S_{\text{past}}] = \mathbf{Pr}[\mathcal{E}_{>\ell} \mid S_{\text{past}}]$, which will complete the proof. As in the setup of Lemma 4.13, we can think of the algorithm as running on bit streams $B_{>\ell}$ and $B_{\leq\ell}$. Since $\mathcal{E}_{>\ell}$ depends only on S_{past}, the adversarial updates $\mathcal{A}$ and above-ℓ changes, we know from Lemma 4.13 that whether or not $\mathcal{E}_{>\ell}$ is true depends only on S_{past}, $\mathcal{A}$ and the bits in $B_{>\ell}$. We will next show that Y_i only depends on bits of $B_{\leq\ell}$ that are independent from S_{past}. All the bits in $B_{\leq\ell}$ are by definition independent from $B_{>\ell}$, and, by the obliviousness of the adversary, from $\mathcal{A}$. Thus, Y_i is an event that depends only on random bits that are independent from the random bits/events that $\mathcal{E}_{>\ell}$ depends on, so we have $\mathbf{Pr}[\mathcal{E}_{>\ell} \mid Y_i, S_{\text{past}}] = \mathbf{Pr}[\mathcal{E}_{>\ell} \mid S_{\text{past}}]$, as desired.

It remains to show that Y_i only depends on bits of $B_{\leq\ell}$ that are independent from S_{past}. This follows from the fact that Y_i is determined by the call to RandomSettle(x, ℓ) made after all the changes in S_{past} have already occurred. Because the call is at level ℓ, the bits come from $B_{\leq\ell}$; because the call uses fresh randomness, these bits are independent from S_{past}. □

4.5.3 Proof of Lemma 4.8. With the hierarchical independence in place, we are now ready to begin the proof of Lemma 4.8. We will actually prove a slightly stronger statement, which shows that only the bits in $B_{\leq\ell}$ need to be random for the lemma to hold; the bits in B^+ can be chosen adversarially.

Lemma 4.23 (Stronger Version of Lemma 4.8). *Let $0 \leq \ell \leq \lfloor \log_4(n) \rfloor$ be any level in the hierarchy. Let $\mathcal{A}$ be the sequence of adversarial updates, and fix* any *bit stream B^+. Consider running our dynamic matching algorithm with $B_{>\ell}$ set to B^+. Let (x, y) be any edge at any time t^* during the update sequence. Then:* $\mathbf{Pr}$[*at time t^*, (x, y) is a matching edge at level ℓ and* responsible(x) *is True*] $= O(\log^3(n)/4^{\ell})$*, where the probability is over all random bits in $B_{\leq\ell}$.*

We will proceed as follows: We will define a special type of hierarchy change, called pivotal change, and will first show (Lemma 4.32) that if a pivotal change exists before time t^*, then the desired statement of Lemma 4.23 holds. Next (Lemma 4.35), we show that a pivotal change exists before time t^* with high probability. To do that we focus our attention on relevant hierarchy changes, which are formalized in the definitions of (ℓ, v)-critical changes and (ℓ, v)-reset-opportunities below.

For the rest of this section, we set $\alpha_\ell = 4000 \cdot \log(n) \cdot 4^\ell$ and $\beta_\ell = \alpha/4 = 1000 \cdot \log(n) \cdot 4^\ell$.

Definition 4.24. Let x, y, t^*, ℓ, B^+ be the variables from the statement of Lemma 4.23. Consider some execution of the algorithm with $B_{>\ell}$ set to B^+. Let $\mathcal{E}^{\text{lemma}}$ refer to the property that (x, y) is a matching edge at level ℓ and that RESPONSIBLE(x) is true and let $\mathcal{E}^{\text{lemma}}_{t^*}$ be the event that $\mathcal{E}^{\text{lemma}}$ holds at time t^*. Note that Lemma 4.23 is equivalent to the statement that $\mathbf{Pr}[\mathcal{E}^{\text{lemma}}_{t^*}] = O(\log^3(n)/4^\ell)$, where the probability is over all random bits in $B_{\leq \ell}$.

To characterize which changes in the hierarchy can lead to changes in the matching, we introduce the following definition:

Definition 4.25. For any vertex v and any level ℓ, we say that a hierarchy change σ is an (ℓ, v)-critical change if σ is a change above ℓ *and* one of the following holds:

(1) σ changes LEVEL(v).
(2) For some neighbor w of v, σ changes LEVEL(w) from $\ell + 1$ to ℓ.
(3) For some neighbor w of v, σ raises w from a level $\leq \ell$ to a level $> \ell$.
(4) σ is an adversarial insertion/deletion of some edge (v, w) incident to v (regardless of level).

We need this definition for the following reason: Only a (LEVEL$(v), v$)-critical change can make the matched edge (v, u) with responsible v unmatched, as shown in the next lemma.

LEMMA 4.26. *A matched edge (u, v) with responsible v becomes unmatched only after a* (LEVEL$(v), v$)*-critical change.*

PROOF. A matched edge (u, v) with responsible v becomes unmatched only if (a) it is deleted, (b) one of its endpoints u or v moves to a higher level, or (c) a RESETMATCHING(v) was executed. Note that a RESETMATCHING(u) would have no effect, as u is not responsible for the matched edge.

Furthermore, RESETMATCHING(v) is only called if a neighbor of v falls from level LEVEL$(v) + 1$ to level LEVEL(v) or an edge incident to v is inserted.

We can summarize this as follows: A matched edge (u, v) with responsible v becomes unmatched only if (a) an edge incident to v is inserted or if the matched edge incident to v is deleted, (b) either u or v moves to a higher level, or (c) a neighbor of v falls from level LEVEL$(v) + 1$ to level LEVEL(v). Note that all of these changes are (LEVEL$(v), v$)-critical. Thus, a matched edge (u, v) with responsible v becomes unmatched only after a (LEVEL$(v), v$)-critical change. □

We also need to characterize after what type of hierarchy changes a RESETMATCHING-operation can be executed. This is the reason for the following definition:

Definition 4.27. For any vertex v and level ℓ, we say that a hierarchy change σ is an (ℓ, v)-reset-opportunity if one of the following holds:

(1) For some neighbor w of v, σ changes LEVEL(w) from $\ell + 1$ to ℓ.
(2) σ is an adversarial insertion of some edge (v, w) incident to v (regardless of level).

LEMMA 4.28. *A* RESETMATCHING(v) *operation is executed only after a* (LEVEL$(v), v$)*-reset opportunity.*

Proof. The operation ResetMatching(v) is called either after an adversarial insertion of an edge incident to v or if a neighbor of v drops from level level(v) + 1 to level(v). □

Observation 4.29. *If v is at level ℓ with* responsible(v) *set to True, and the algorithm performs a change σ that is a (ℓ, v)-reset opportunity, then with probability $p_\ell^{\text{reset}} = 1/4^{\ell+3}$ the algorithm does* ResetMatching(v). *(See Line 51 of Algorithm 2 and Line 16 of Algorithm 1.)*

4.5.4 Intuition for the Proof of Lemma 4.23. With all our definitions in place, we now give a brief intuition for the proof of Lemma 4.23. Let $\sigma_1, \ldots, \sigma_{q^*}$ be all the hierarchy changes performed by the algorithm up to time t^*. Recall that we want to bound the probability that $\mathcal{E}^{\text{lemma}}$ holds after change σ_{q^*}. Let us focus on any change σ_i, and consider two cases.

Case 1: There are many (ℓ, x)-critical changes between σ_i and σ_{q^*}. Note that whether or not we fall in Case 1 is independent of any random choices made after σ_i (recall that only bits from $B_{\leq \ell}$ are random), because these (ℓ, x)-critical changes are by definition changes above ℓ, so we can apply Corollary 4.22. We will show that in Case 1, either one of these critical changes causes x to change level (and hence to pick a new mate), or they will cause so many (ℓ, x) reset-opportunities that by Observation 4.29 there is a high probability that the algorithm will perform a ResetMatching(x) before change σ_{q^*}. Either way, x will pick a new random mate at some point between $\sigma_i, \ldots, \sigma_{q^*}$, so the matching at change σ_i bears no relevance to the matching at change σ_{q^*}. Any σ_i in Case 1 can thus be effectively ignored.

Case 2: There are few (ℓ, x)-critical changes between σ_i and σ_{q^*}. For simplicity, let us assume that none of these (ℓ, x)-critical changes change the level of x, and when σ_i is performed, $\mathcal{E}^{\text{lemma}}$ is false. For $\mathcal{E}^{\text{lemma}}$ to become true, some hierarchy change between σ_i and σ_{q^*} must choose (x, y) as the matching. By Matching Property*, we know that any one particular RandomSettle(x, ℓ) has only a small chance of picking (x, y); to complete the proof, we will show that because we are in Case 2, there are (with high probability) few executions of RandomSettle(x, ℓ) between σ_i and σ_{q^*}. The reason there are few executions is that if RandomSettle(x, ℓ) picks some matching edge (x, z), then the only way the algorithm calls a new RandomSettle(x, ℓ) is if the edge (x, z) is removed from the matching due to an adversarial deletion or a hierarchy change. We will show that this can only occur due to a (ℓ, x)-critical change, and that moreover, any (ℓ, x)-critical change is unlikely to affect (x, z), because z was chosen at random from among many choices, and the (ℓ, x)-critical changes are independent of this random choice (Corollary 4.22). Since there are few (ℓ, x)-critical changes remaining (because we are in Case 2), they are unlikely to cause many executions of RandomSettle(x, ℓ).

Formalizing the Proof. We now define the notion of a pivotal change, which corresponds to a hierarchy change that satisfies the assumptions of Case 2 in the above intuition.

Definition 4.30. Define x, y, ℓ, B^+, t^* as in the statement of Lemma 4.23. Let $\sigma_1, \ldots, \sigma_{q^*}$ be the sequence of hierarchy changes up to time t^*. Consider some execution of the dynamic matching algorithm with $B_{>\ell}$ set to B^+. We say that some change σ performed by the algorithm is *pivotal for time t^** if it satisfies *all* of the following properties:

(1) The number of (ℓ, x)-critical changes between σ and σ_{q^*} is at most α_ℓ. (Recall that $\alpha_\ell = 4000 \cdot \log n \cdot 4^\ell$.)
(2) There are no (ℓ, x)-critical changes between σ and σ_{q^*} that alter the level of x. (Note that x may still move levels due to hierarchy changes that are not above ℓ.)
(3) $\mathcal{E}^{\text{lemma}}$ is false right after change σ.

Technical note: The lemmas and definitions are made cleaner if at the very beginning of the algorithm (when the graph is still empty) we insert a dummy update σ^{dummy} that does nothing; this is solely to allow for the possibility that the so-to-speak 0th update σ^{dummy} is itself pivotal.

The crucial property of a change σ that is pivotal for time t^* is as follows: Whether σ is pivotal for t^* or not only depends on (i) the hierarchy at the time of change σ, (ii) the above-ℓ changes between σ and time t^*, and (iii) adversarial updates.

Lemma 4.31. *Consider a change $\sigma = \sigma_i$ with $i < t^*$. Whether σ is pivotal for time t^* is uniquely determined by (i) the hierarchy at the time of change σ including the* responsible *bits, (ii) the above-ℓ changes between σ and time t^*, and (iii) adversarial updates.*

Proof. We will show that each of the properties of a pivotal change depend only on the hierarchy at the time of change σ, the above-ℓ changes between σ and time t^*, and the adversarial updates.

For Property (1), note that whether a change is (ℓ, x)-critical only depends on whether the change is above ℓ and whose node's level it changes (if any), and whether it is an adversarial update. All this can be determined for a change up to time t^* if the information in (i)–(iii) is known. Thus, with information (i)–(iii) it is uniquely determined whether there are at most α_ℓ (ℓ, x)-critical changes between σ and σ_{q^*}, i.e., whether Property (1) holds.

Property (2) guarantees that any (ℓ, x)-critical changes that occur after σ and up to time t^* *must* be of type (2)–(4) of the definition of a critical change. As for Property (1) whether a change is (ℓ, x)-critical and whether it changes the level of x is uniquely determined by information (i)–(iii).

Property (3) of a pivotal change, i.e., whether $\mathcal{E}^{\text{lemma}}$ is false right after σ is uniquely determined by the hierarchy right before change σ including the responsible bits as well as on σ. □

Note that it follows from the lemma that whether a change σ_i with $i < t^*$ is pivotal depends only on information (i)—(iii) from the lemma. We now proceed as follows: Lemma 4.32 will show that Lemma 4.23 holds if we condition on the existence of a pivotal change for t^*; this corresponds to Case 2 in the intuition section above. Lemma 4.35 will then show that a pivotal change exists with high probability; this corresponds to the intuition above that Case 1 can be effectively ignored, and only Case 2 is relevant.

Lemma 4.32. *Define x, y, ℓ, B^+, t^* as in the statement of Lemma 4.23. Consider some execution of the dynamic matching algorithm with $B_{>\ell}$ set to B^+, and condition on the fact that during the execution of the algorithm up to time t^* the algorithm encounters a change σ^{pivot} that is pivotal for time t^*. Then,* $\mathbf{Pr}[$*at time t^*, (x, y) is a matching edge that was chosen by some* RandomSettle$(x, \ell)]$ $= O(\log^3(n)/4^\ell)$*, where the probability is over all random bits in $B_{\le \ell}$.*

Proof. Let $\sigma'_1, \ldots, \sigma'_\alpha$ be the sequence of (ℓ, x)-critical changes between change σ^{pivot} and time t^*. By definition of a pivotal change for t^*, we have that $\alpha \le \alpha_\ell$, and that none of the σ'_i change the level of x.

Recall that by definition of a pivotal change, $\mathcal{E}^{\text{lemma}}$ is false after change σ^{pivot}. Thus, for $\mathcal{E}^{\text{lemma}}$ to become true, at some point between σ^{pivot} and t^* the algorithm must call RandomSettle(x, ℓ), and this call must choose the particular edge (x, y) as the new matching edge. Let X^{settle} be the random variable that is equal to the number of times we call RandomSettle(x, ℓ) between σ^{pivot} and t^* and let $\mathcal{E}^{\text{settle}}$ be the event that $X^{\text{settle}} \le C' \log^2(n)$ for some large constant C'. The crux of the proof is to show that

$$\mathbf{Pr}[\mathcal{E}^{\text{settle}}] \ge 1 - 1/n^5. \tag{7}$$

Before proving this fact, let us show why it allows us to prove the lemma. We have that

$$\mathbf{Pr}[\mathcal{E}^{\text{lemma}}_{t^*}] = \mathbf{Pr}[\mathcal{E}^{\text{lemma}}_{t^*} \wedge \mathcal{E}^{\text{settle}}] + \mathbf{Pr}[\mathcal{E}^{\text{lemma}}_{t^*} \wedge \neg\mathcal{E}^{\text{settle}}] \leq \mathbf{Pr}[\mathcal{E}^{\text{lemma}}_{t^*} \wedge \mathcal{E}^{\text{settle}}] + \frac{1}{n^5}. \quad (8)$$

We now bound $\mathbf{Pr}[\mathcal{E}^{\text{lemma}}_{t^*} \wedge \mathcal{E}^{\text{settle}}]$. By Matching Property*, an execution of RandomSettle(x, ℓ) has probability at most $O(\log(n)/4^\ell)$ of picking edge (x, y). However, we need a bound on the probability of (x, y) being matched *at time* t^*. This is where we rely on the independence proved in Corollary 4.22: Any call to RandomSettle(x, ℓ) between σ^{pivot} and t^* (a) uses only bits from $B_{\leq\ell}$ and (b) depends only on which neighbors belong to $N_{<\ell}(x)$ at the time of the call. Note that (a) the used bits are fresh and (b) which neighbors belong to $N_{<\ell}(x)$ depends only on S_{past}, i.e., the sequence of hierarchy changes before this call to RandomSettle(x, ℓ). As shown in Lemma 4.31 whether or not σ^{pivot} is pivotal for time t^* depends only on the hierarchy at change σ^{pivot} (i.e., S_{past}), future above-ℓ changes, and $\mathcal{A}$. Thus, it is an event that fulfills the requirements in Corollary 4.22, which shows that mate(x) is chosen uniformly at random from $N_{<\ell}(x)$.

Now, by definition of $\mathcal{E}^{\text{settle}}$ there are $O(\log^2(n))$ executions of RandomSettle(x, ℓ) between σ^{pivot} and t^*. Each of these picks (x, y) with probability $O(\log(n)/4^\ell)$. Thus, by a union bound,

$$\mathbf{Pr}[\mathcal{E}^{\text{lemma}}_{t^*} \wedge \mathcal{E}^{\text{settle}}] \leq \mathbf{Pr}[\mathcal{E}^{\text{lemma}}_{t^*} \mid \mathcal{E}^{\text{settle}}] = O\left(\log^2(n) \cdot \frac{\log(n)}{4^\ell}\right) = O\left(\frac{\log^3(n)}{4^\ell}\right). \quad (9)$$

Combining the three equations above completes the proof of the lemma. Thus, all that is left to do is to prove Equation (7).

Proof of Equation (7). Recall that $\mathcal{E}^{\text{settle}}$ only considers the time period between σ^{pivot} and t^*, and that $\sigma'_1, \ldots, \sigma'_\alpha$ is the sequence of (ℓ, x)-critical changes in this time period, with $\alpha \leq \alpha_\ell$. Consider some call to RandomSettle(x, ℓ) during this time period, which results in some edge e being chosen as the matching edge and responsible(x) being set to True. The only way that RandomSettle(x, ℓ) can be called again after this point is that edge e leaves the matching. By Lemma 4.26 this can only happen as the consequence of an (ℓ, x)-critical change. Since by the definition of σ^{pivot}, none of the (ℓ, x)-critical changes σ'_i alter the level of x, only (ℓ, x)-critical changes of types 2, 3, and 4 from Definition 4.25 are possible in the sequence $\sigma'_1, \ldots, \sigma'_\alpha$, i.e., the following types of changes:

- For some neighbor w of x, σ changes level(w) from $\ell + 1$ to ℓ.
- For some neighbor w of x, σ raises w from a level $\leq \ell$ to a level $> \ell$.
- σ is an adversarial insertion/deletion of some edge (x, w) incident to x (regardless of level).

We say the corresponding neighbor w in such a change is the node *affected* by the change.

In the following, we distinguish between calls to RandomSettle(x, ℓ) that are the consequence of an (ℓ, x)-reset opportunity and those that are not and in the latter case, we consider affected node to the call to RandomSettle(x, ℓ).

Bounding X^{settle}. Recall that X^{settle} is the random variable that is equal to the number of times the algorithm calls RandomSettle(x, ℓ) between σ^{pivot} and t^*. Consider the sequence of the algorithm's calls to RandomSettle(x, ℓ) and ResetMatching(x) between σ^{pivot} and t^*. Let X^{reset} be the random variable that is equal to the number of calls to RandomSettle(x, ℓ) that are directly preceded by a call to ResetMatching(x) in this sequence and let X^{crs} be the random variable that is equal to the number of calls to RandomSettle(x, ℓ) that are directly preceded by a call to RandomSettle(x, ℓ) in this sequence. Observe that trivially

$$X^{\text{settle}} \leq X^{\text{reset}} + X^{\text{crs}} + 1\,. \quad (10)$$

(The plus one is necessary, because the sequence of calls to RandomSettle(x,ℓ) and ResetMatching(x) might start with a call to RandomSettle(x,ℓ) for which then no preceding call exists.)

Bounding X^{reset}. We will first bound X^{reset} by bounding the corresponding number of calls to ResetMatching(x). For each call to ResetMatching(x) preceding a call to RandomSettle(x,ℓ), we necessarily have level$(x) = \ell$ and therefore there must be some (ℓ,x)-reset opportunity σ'_i causing the call to ResetMatching(x). Each (ℓ,x)-reset opportunity has a probability of at most $p_\ell^{\text{reset}} = 1/4^{\ell+3}$ of causing a ResetMatching(x). By definition of σ^{pivot}, there are at most $\alpha_\ell = 4,000 \cdot \log(n) \cdot 4^\ell$ (ℓ,x)-critical changes in the sequence $\sigma'_1,\ldots,\sigma'_\alpha$. Thus, applying Chernoff bound and a suitable choice of constants, with probability at least $1 - 1/(2n^5)$, we have that $X^{\text{reset}} = O(\log(n))$.

Bounding X^{crs}. We will now bound X^{crs}. In particular, we will show that with high probability $X^{\text{crs}} < k := 1024000C\log^2(n)$. Note that X^{crs} is equal to the number of pairs of consecutive calls to RandomSettle(x,ℓ) between σ^{pivot} and t^* that are not interleaved with calls to ResetMatching(x) Consider a pair of consecutive calls to RandomSettle(x,ℓ) that are not interleaved with calls to ResetMatching(x). By Matching Property*, we have $|N_{<\ell}(x)| \geq \frac{4^\ell}{32C\log(n)}$ during both these calls to RandomSettle(x,ℓ). Define $\gamma := \frac{4^\ell}{32C\log(n)}$ to be this lower bound on the number of choices for the mate of x. Let $y_1, y_2, \ldots$ be the sequence of nodes from $N_{<\ell}(x)$ appearing as affected nodes in those (ℓ,x)-critical changes that are not reset opportunities after the first call to RandomSettle(x,ℓ) in the order of first appearance. We say that the call to RandomSettle(x,ℓ) has *large span* if mate(x) does not occur within the first (up to) $\frac{\gamma}{2}$ elements of this sequence $y_1, y_2, \ldots$. As none of the reset opportunities after the first call is successful (as there is no ResetMatching(x) before the second RandomSettle(x,ℓ)), the fact that the first call has a large span implies that there are at least $\gamma/2$ (ℓ,x)-critical changes that are not reset opportunities between the two calls.

Claim 4.33. Each call to RandomSettle(x,ℓ) has large span with probability at least $\frac{1}{2}$ and this bounds holds independently of which earlier calls had large span.

Proof. Consider a call to RandomSettle(x,ℓ) and let $N := N_{<\ell}(x)$ when this call happens. As a result of this call, the mate mate(x) of x is chosen uniformly at random from N with $|N| \geq \gamma$. Let $y_1,\ldots,y_{\gamma'}$ be the set of nodes from $N_{<\ell}(x)$ appearing as affected nodes in those (ℓ,x)-critical changes that are not reset opportunities after the call to RandomSettle(x,ℓ) in the order of first appearance.

Note that N only depends on the sequence of hierarchy changes before the call to RandomSettle(x,ℓ), and the sequence $y_1,\ldots,y_{\gamma'}$ only depends on the sequence of hierarchy changes before the call and the above-ℓ hierarchy changes after the call. Therefore, we may apply Corollary 4.22 by which the probability that a specific y_i is equal to mate(x) is at most $\frac{1}{\gamma}$. Note that this holds for any possible S_{past}, and, thus, independent of S_{past}. Hence, it holds in particular no matter which previous calls to RandomSettle(x,ℓ) had large span. It follows that mate(x) occurs within the first (up to) $\frac{\gamma}{2}$ elements of the sequence $y_1,\ldots,y_{\gamma'}$ with probability at most $\frac{\gamma}{2}\cdot\frac{1}{\gamma} = \frac{1}{2}$. This means that each call to RandomSettle(x,ℓ) has large span with probability at least $\frac{1}{2}$.

Note that this reasoning also shows that whether a call has large span is independent of whether a *previous* call had large span. □

We next bound how often calls with large span can happen for consecutive calls to RandomSettle(x,ℓ).

Claim 4.34. There are at most $\frac{1}{4}k$ pairs of consecutive calls to RandomSettle(x,ℓ) between σ^{pivot} and t^* that are not interleaved with any call to ResetMatching(x) such that the first of these calls has large span.

Proof. Consider any pair of consecutive calls to RandomSettle(x, ℓ) between σ^{pivot} and t^* that are not interleaved with any call to ResetMatching(x). If the first of these two calls has large span, then there are at least $\frac{\gamma}{2} = \frac{4^\ell}{64C\log(n)}$ (ℓ, x)-critical changes that are not reset opportunities between these two calls, as argued above. Since the total number of (ℓ, x)-critical changes that are not reset opportunities between σ^{pivot} and t^* is at most $\alpha_\ell = 4,000 \cdot \log(n) \cdot 4^\ell$, it must be the case that the situation above occurs at most $\frac{\alpha_\ell}{\gamma/2} = 256,000C\log^2(n) = \frac{1}{4}k$ times. □

Recall that X^{crs} is the total number of pairs of consecutive calls to RandomSettle(x, ℓ) between σ^{pivot} and t^* that are not interleaved with any call to ResetMatching(x). The claim together with the fact that each call to RandomSettle(x, ℓ) has large span with probability at least $\frac{1}{2}$ gives us a bound on X^{crs} as follows: For each $i \geq 1$, define Z_i as the binary random variable that (1) if $i \leq X^{\text{crs}}$ and in the ith pair of consecutive calls to RandomSettle(x, ℓ) the first call to RandomSettle(x, ℓ) has large span is 1, (2) if $i \leq X^{\text{crs}}$ and in the ith pair of consecutive calls to RandomSettle(x, ℓ) the first call to RandomSettle(x, ℓ) does not have large span is 0, and (3) if $i > X^{\text{crs}}$ is 1 with probability $\frac{1}{2}$ and 0 otherwise. Furthermore, define, for each $i \geq 1$, Z_i^* as the binary random variable that is 1 with probability $\frac{1}{2}$ and 0 otherwise.

We will now show that $\Pr[X^{\text{crs}} \geq k] \leq \Pr[\sum_{i=1}^k Z_i \leq \frac{1}{4}k]$ by arguing that the event $X^{\text{crs}} \geq k$ implies the event $\sum_{i=1}^k Z_i \leq \frac{1}{4}k$. Observe that this implication is equivalent to the statement $\Pr[\sum_{i=1}^k Z_i \leq \frac{1}{4}k \mid X^{\text{crs}} \geq k] = 1$. Given that $X^{\text{crs}} \geq k$, it follows from the definition above that for each $i \leq k$, Z_i is 1 if in the ith pair of consecutive calls to RandomSettle(x, ℓ) the first call to RandomSettle(x, ℓ) had large span. In other words, conditioned on the event $X^{\text{crs}} \geq k$, $\sum_{i=1}^k Z_i$ is precisely the number of pairs of consecutive calls to RandomSettle(x, ℓ) between σ^{pivot} and t^* that are not interleaved with any call to ResetMatching(x) and in which the first call to RandomSettle(x, ℓ) has large span. By Claim 4.34, we have that this number is at most $\frac{1}{4}k$, i.e., conditioned on $X^{\text{crs}} \geq k$, we have $\sum_{i=1}^k Z_i \leq \frac{1}{4}k$ as desired.

Now by Claim 4.33 each Z_i with $i \leq X^{\text{crs}}$ is 1 with probability at least $\frac{1}{2}$ *independent* of the outcomes of the random variables with smaller index, and for $i > X^{\text{crs}}$ this property obviously holds as well. More formally, the following holds for all $z_1, \ldots z_{i-1} \in \{0, 1\}$:

$$\mathbf{Pr}[Z_i = 1 \mid Z_1 = z_1, \ldots, Z_{i-1} = z_{i-1}] \geq \frac{1}{2} = \mathbf{Pr}[Z_i^* = 1] .$$

Under this precondition, the sum of the Z_i's stochastically dominates the sum of the Z_i^*'s (see Lemma 1.8.7 in Reference [20]), i.e., $\mathbf{Pr}[\sum_{i=1}^k Z_i \leq \lambda] \leq \mathbf{Pr}[\sum_{i=1}^k Z_i^* \leq \lambda]$ for all λ. Using the shorthand $\mu := \mathbf{E}[\sum_{i=1}^k Z_i^*] = \frac{1}{2}k$, we now apply a standard Chernoff bound to get the following estimation:

$$\begin{aligned}\mathbf{Pr}[X^{\text{crs}} \geq k] &\leq \mathbf{Pr}\left[\sum_{i=1}^k Z_i \leq \frac{1}{4}k\right] = \mathbf{Pr}\left[\sum_{i=1}^k Z_i \leq \frac{1}{2}\mu\right] \\ &\leq \mathbf{Pr}\left[\sum_{i=1}^k Z_i^* \leq \frac{1}{2}\mu\right] \leq \exp\left(-\frac{1}{8}\mu\right) = \exp\left(-\frac{1}{16}k\right) \leq \frac{1}{2n^5} .\end{aligned}$$

Overall, we have thus argued that with probability at least $1 - \frac{1}{2n^5}$ both X^{reset} and X^{crs}, and thus X^{settle} by Equation (10), are at most $O(\log^2 n)$. □

Lemma 4.35. *Define x, y, ℓ, B^+, t^* as in the statement of Lemma 4.23. Consider some execution of the dynamic matching algorithm with $B^+ = B_{>\ell}$. Then,* $\mathbf{Pr}$[*at some point during the execution the algorithm encounters a pivotal change σ^{pivot} for time t^**] $\geq 1 - 1/n^{10}$*, where the probability is over all random bits in $B_{\leq \ell}$.*

Proof. Let $\sigma_1^\ell, \ldots \sigma_q^\ell$ be all the (ℓ, x)-critical changes made by the algorithm up to time t^*. We next define two (ℓ, x)-critical changes and argue about their relative order. (a) If $q \leq \alpha_\ell$, then we set $\sigma^{\text{gap}} = \sigma^{\text{dummy}}$, where σ^{dummy} is the empty update defined in Definition 4.30. If $q > \alpha_\ell$, let σ^{gap} be the (ℓ, x)-critical change $\sigma^\ell_{q-\alpha_\ell}$; note that this is chosen to satisfy Property 1 of the definition of a pivotal change (Definition 4.30). (b) Let σ^{level} be the last (ℓ, x)-critical change that changes $\text{LEVEL}(x)$ and $\sigma^{\text{level}} = \sigma^{\text{dummy}}$ if no such change exists. It is clear that σ^{level} satisfies property 2 of the definition of a pivotal change for t^*. Observe that σ^{level} also satisfies property 3, because our algorithm only alters the level of a *free* vertex, so x is free right after σ^{level}, which implies that $\mathcal{E}^{\text{lemma}}$ is false at that time. (This last argument might feel like cheating, since soon after σ^{level} the algorithm will assign a new matching edge to x; but these future matching edges were already handled in Lemma 4.32, where we bounded the probability that $\mathcal{E}^{\text{lemma}}$ becomes true at some point before t^*.)

We now consider two cases. The simple case is that σ^{level} comes after, or is the same as, σ^{gap}. In this case, σ^{level} satisfies all the properties of a pivotal change for t^*, thus proving the lemma.

The second case is that σ^{gap} comes after σ^{level}. In this case σ^{gap} satisfies Properties 1 and 2, but may fail to satisfy property 3. If $\sigma^{\text{gap}} = \sigma^{\text{dummy}}$, then, since $\mathcal{E}^{\text{lemma}}$ is clearly false after σ^{dummy} (the graph is still empty), σ^{dummy} is itself pivotal for t^*. Also note that $\mathcal{E}^{\text{lemma}}$ is false before and after σ^{gap} if $\text{LEVEL}(x) \neq \ell$ at time σ^{gap}.

Assumption: We can thus assume for the rest of the proof that $\sigma^{\text{gap}} \neq \sigma^{\text{dummy}}$, that $\text{LEVEL}(x) = \ell$ right before and right after σ^{gap} and x remains on this level until t^*.

Note that if there exists *any* change, let us call it σ^{pivot}, between σ^{gap} and t^* such that $\mathcal{E}^{\text{lemma}}$ is false after change σ^{pivot}, then σ^{pivot} satisfies all the properties of a pivotal change for t^*, as all changes after change σ^{gap} fulfill properties 1 and 2 of a pivotal change for t^*. Let $\mathcal{E}^{\text{pivot}}$ be the event that such a σ^{pivot} exists; we now show that $\mathcal{E}^{\text{pivot}}$ is true with high probability. The crux of our argument is to show that there are many (ℓ, x)-reset-opportunities between σ^{gap} and t^*: Each of them performs a $\text{ResetMatching}(x)$ only with a small probability, but if there are enough of them at least one of them will indeed perform a $\text{ResetMatching}(x)$, which will imply that x becomes free, i.e., Property 3 holds after this change, i.e., it is a pivotal change for t^*. Thus, we first need to show the following claim. Recall that $\beta_\ell = \alpha_\ell/4$. Recall that we are in the case $\sigma^{\text{gap}} > \sigma^{\text{level}}$, which implies that there are exactly α_ℓ (ℓ, x)-critical changes between σ^{gap} and t^*.

Claim 4.36. If $\sigma^{\text{gap}} > \sigma^{\text{level}}$, then at least β_ℓ of the α_ℓ (ℓ, x)-critical changes between σ^{gap} and t^* are also (ℓ, x)-reset-opportunities.

Proof of Claim. Because $\sigma^{\text{gap}} > \sigma^{\text{level}}$, we know that none of the (ℓ, x)-critical changes between σ^{gap} and t^* change the level of x. Thus, each of these critical changes either: **(1)** adds a vertex w to $N_{=\ell}(x)$ by moving w from level $\ell + 1$ to level ℓ (item 2 of Definition 4.25), or **(2)** removes a vertex w from $N_{\leq \ell}(x)$ (item 3), or **(3)** it makes an adversarial update incident to x (item 4). Let t_i be the number of changes of each type. Note that $t_1 + t_2 + t_3 = \alpha_\ell$. Types 1 and 3 are (ℓ, x) reset opportunities by definition. Now, note that at the start of σ^{gap}, since $\text{LEVEL}(x) = \ell$ (see assumption above), Invariant 3 guarantees that $|N_{\leq \ell}(x)| = N_{<\ell+1}(x) \leq 4^{\ell+1}$. Moreover, $N_{\leq \ell}$ can only grow as a result of changes of types 1 and 3, and only by 1 after each such change. Thus, $t_2 \leq 4^{\ell+1} + t_1 + t_3$. Thus, it follows that [# reset opportunities] $= t_1 + t_3 \geq (\alpha_\ell - 4^{\ell+1})/2 \geq \alpha_\ell/4 = \beta_\ell$. □

Back to Proof of Lemma 4.35. Let $\sigma^*_1, \ldots, \sigma^*_{\beta_\ell}$ be (ℓ, x)-critical changes between σ^{gap} and t^* that are also (ℓ, x)-reset opportunities; these changes exist by the claim above (there may be more than β_ℓ such changes, in which case pick any β_ℓ). Our goal is to show that $\mathcal{E}^{\text{lemma}}$ is false after one of these changes whp, which implies that $\mathcal{E}^{\text{pivot}}$ holds whp.

Let $\mathcal{E}_i^{\text{no-pivot}}$ be the event that all of the following hold right before change σ_i^*: RESPONSIBLE(x) is True and LEVEL$(x) = \ell$ and the change σ_i^* does not lead to RESETMATCHING(x). Recall that by our assumption, we know that LEVEL$(x) = \ell$ for each of the σ_i^* that we are considering. We need to show that with high probability after at least one of the changes σ_i^* $\mathcal{E}^{\text{lemma}}$ is false, i.e., RESPONSIBLE(x) = False or RESETMATCHING (x) is executed in the change. Event $\mathcal{E}^{\text{pivot}}$ is false if no such change exists. Thus, $\mathcal{E}^{\text{pivot}}$ can only be false if *all* the $\mathcal{E}_i^{\text{no-pivot}}$ are true. We thus have

$$\mathbf{Pr}[\neg\mathcal{E}^{\text{pivot}}] \leq \mathbf{Pr}[\mathcal{E}_1^{\text{no-pivot}} \wedge \ldots \wedge \mathcal{E}_{\beta_\ell}^{\text{no-pivot}}] \leq \mathcal{E}_1^{\text{no-pivot}} \cdot \prod_{i=2}^{\beta_\ell} \mathbf{Pr}[\mathcal{E}_i^{\text{no-pivot}} \mid \mathcal{E}_1^{\text{no-pivot}} \wedge \ldots \wedge \mathcal{E}_{i-1}^{\text{no-pivot}}].$$

We now observe that $\mathbf{Pr}[\mathcal{E}_1^{\text{no-pivot}}] \leq (1 - p_\ell^{\text{reset}}) = (1 - 1/4^{\ell+3})$ and $\mathbf{Pr}[\mathcal{E}_i^{\text{no-pivot}} \mid \mathcal{E}_1^{\text{no-pivot}} \wedge \ldots \wedge \mathcal{E}_{i-1}^{\text{no-pivot}}] \leq (1 - p_\ell^{\text{reset}}) = (1 - 1/4^{\ell+3})$. The reason is simply that each σ_i^* is a (ℓ, x) reset-opportunity, so either RESPONSIBLE(x) is False at change σ_i^*, in which case $\mathcal{E}_i^{\text{no-pivot}}$ is false by definition, or otherwise by Observation 4.29 the algorithm performs RESETMATCHING (x) with probability p_ℓ^{reset}. Moreover, the probability of RESETMATCHING(x) occurring is using a fresh random bit and, thus, is clearly independent of everything that came before.

Putting everything together, we have that

$$\mathbf{Pr}[\neg\mathcal{E}^{\text{pivot}}] \leq \left(1 - \frac{1}{4^{\ell+3}}\right)^{\beta_\ell} = \left(1 - \frac{1}{64 \cdot 4^\ell}\right)^{1000 \cdot \log(n) \cdot 4^\ell} \leq \frac{1}{n^{10}}.$$

Note that this crucially relies on Corollary 4.22 to ensure that the probability of RESETMATCHING(x) is not correlated with any of the future (ℓ, x)-critical changes. □

PROOF OF LEMMA 4.23. The proof follows immediately from the combination of Lemmas 4.35 and 4.32 □

4.5.5 Putting Everything Together. We are now ready to prove that the algorithm processes *every* adversarial update in expected time $O(\log^4(n))$.

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 $O_u, O_v, \mathcal{E}_u^{\text{LEVEL}(v)}, \mathcal{E}_v^{\text{LEVEL}(u)}$ in $O(1)$ time, and then it has to perform $O(\log(n))$ calls to CHECKFORRISE(u, i) and CHECKFORRISE(v, i) (Line 14 or Algorithm 1). Each CHECKFORRISE(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 4.5), so plugging in the cost of a call to RISE from Lemma 4.4, the expected total time to process the rises is $O(\sum_{i=0}^{\lfloor \log_4(n) \rfloor} (\log(n)/4^i) \cdot 4^i = O(\log^2(n))$. INSERT (x, y) also causes RESETMATCHING(u) and RESETMATCHING(v) with Probabilities $p^{\text{reset}}_{\text{LEVEL}(u)} = 1/4^{\text{LEVEL}(u)+3}$ and $p^{\text{reset}}_{\text{LEVEL}(v)} = 1/4^{\text{LEVEL}(v)+3}$. By Lemma 4.6 the expected time to process the resets is $O(4^{\text{LEVEL}(u)})$ and $O(4^{\text{LEVEL}(v)})$; multiplying by the reset probabilities yields an expected time of $O(1)$.

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-ϕ 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, LEVEL(u) = LEVEL(v), so let us say they are both equal to ℓ. The deletion of (u, v) requires the algorithm to execute FIXFREEVERTEX(u) and FIXFREEVERTEX(v), which by Corollary 4.7 requires time $O(4^\ell)$. By Corollary 4.9, the total expected update time is thus $O(\sum_{\ell=0}^{\lfloor \log_4(n) \rfloor} 4^\ell \cdot (\log^3(n)/4^\ell)) = O(\log^4(n))$. □

Explicitly Maintaining a List of the Edges in the Matching. Both the amortized expected algorithm of Baswana et al. [7], 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, then 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 ℓ arbitrary edges of the matching in time $O(\ell)$. The very recent black-box reduction in Reference [38] 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 $O(\log^6(n) + 1/\epsilon)$. The reduction of Reference [38] also applies to the dynamic $(2+\epsilon)$-approximate matching algorithms of Arar et al. [4] and Charikar and Solomon [18], whose update times we can beat for certain regimes of ϵ.

5 CONCLUSION

In this article, we have provided a meta-algorithm that converts dynamic algorithms with a bound on the worst-case expected update time into ones with a high-probability bound on the worst-case time at the expense of logarithmic factors in the update time. We have then applied this reduction to two graph problems: dynamic spanner and dynamic maximal matching. Our main observation for these two problems was that certain deterministic amortization techniques in the known algorithms can be replaced by randomized ones to obtain a worst-case expected bound instead of an amortized one. We conjecture that this approach also works for other graph problems. Additionally it would be interesting to have more meta-theorems for dynamic graph algorithms in addition to sparsification [22] and the expected to high-probability conversion presented here.

ACKNOWLEDGMENTS

The conference version of this article [10] had an error in the analysis of the dynamic matching algorithm. In particular, Lemma 4.5 assumed an independence between adversarial updates to the hierarchy that is in fact true, but which requires a sophisticated proof. We are very grateful to the anonymous reviewers of Transactions on Algorithms for pointing out this mistake in our analysis.

The mistake is fixed in Section 4.5. Almost the entire fix is a matter of analysis: the only change to the algorithm itself is the introduction of responsible bits in Algorithm 2.

The first author would like to thank Mikkel Thorup and Alan Roytman for a very helpful discussion of the proof of Theorem 1.1.

REFERENCES

[1] Amir Abboud and Virginia Vassilevska Williams. 2014. Popular conjectures imply strong lower bounds for dynamic problems. In *Proceedings of the Symposium on Foundations of Computer Science (FOCS)*. 434–443. DOI: https://doi.org/10.1109/FOCS.2014.53

[2] Ittai Abraham, Shiri Chechik, and Sebastian Krinninger. 2017. Fully dynamic all-pairs shortest paths with worst-case update-time revisited. In *Proceedings of the Symposium on Discrete Algorithms (SODA)*. 440–452. DOI: https://doi.org/10.1137/1.9781611974782.28

[3] Ittai Abraham, David Durfee, Ioannis Koutis, Sebastian Krinninger, and Richard Peng. 2016. On fully dynamic graph sparsifiers. In *Proceedings of the Symposium on Foundations of Computer Science (FOCS)*. 335–344. DOI: https://doi.org/10.1109/FOCS.2016.44

[4] Moab Arar, Shiri Chechik, Sarel Cohen, Cliff Stein, and David Wajc. 2018. Dynamic matching: Reducing integral algorithms to approximately-maximal fractional algorithms. In *Proceedings of the International Colloquium on Automata, Languages and Programming (ICALP)*. 7:1–7:16. DOI: https://doi.org/10.4230/LIPIcs.ICALP.2018.7

[5] Giorgio Ausiello, Paolo Giulio Franciosa, and Giuseppe F. Italiano. 2006. Small stretch spanners on dynamic graphs. *J. Graph Algor. Applic.* 10, 2 (2006), 365–385. DOI: https://doi.org/10.7155/jgaa.00133

[6] Baruch Awerbuch. 1985. Complexity of network synchronization. *J. ACM* 32, 4 (1985), 804–823. DOI: https://doi.org/10.1145/4221.4227

[7] Surender Baswana, Manoj Gupta, and Sandeep Sen. 2018. Fully dynamic maximal matching in $O(\log n)$ update time (corrected version). *SIAM J. Comput.* 47, 3 (2018), 617–650. DOI: https://doi.org/10.1137/16M1106158

[8] Surender Baswana, Sumeet Khurana, and Soumojit Sarkar. 2012. Fully dynamic randomized algorithms for graph spanners. *ACM Trans. Algor.* 8, 4 (2012), 35:1–35:51. DOI: https://doi.org/10.1145/2344422.2344425

[9] Soheil Behnezhad, Mahsa Derakhshan, MohammadTaghi Hajiaghayi, Cliff Stein, and Madhu Sudan. 2019. Fully dynamic maximal independent set with polylogarithmic update time. In *Proceedings of the Symposium on Foundations of Computer Science (FOCS)*. 382–405. DOI: https://doi.org/10.1109/FOCS.2019.00032

[10] Aaron Bernstein, Sebastian Forster, and Monika Henzinger. 2019. A deamortization approach for dynamic spanner and dynamic maximal matching. In *Proceedings of the Symposium on Discrete Algorithms (SODA)*. 1899–1918.

[11] Sayan Bhattacharya, Deeparnab Chakrabarty, and Monika Henzinger. 2017. 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)*. 86–98. DOI: https://doi.org/10.1007/978-3-319-59250-3_8

[12] Sayan Bhattacharya, Deeparnab Chakrabarty, Monika Henzinger, and Danupon Nanongkai. 2018. Dynamic algorithms for graph coloring. In *Proceedings of the Symposium on Discrete Algorithms (SODA)*. 1–20. DOI: https://doi.org/10.1137/1.9781611975031.1

[13] Sayan Bhattacharya, Monika Henzinger, and Giuseppe F. Italiano. 2018. Deterministic fully dynamic data structures for vertex cover and matching. *SIAM J. Comput.* 47, 3 (2018), 859–887. DOI: https://doi.org/10.1137/140998925

[14] Sayan Bhattacharya, Monika Henzinger, and Danupon Nanongkai. 2016. New deterministic approximation algorithms for fully dynamic matching. In *Proceedings of the Symposium on Theory of Computing (STOC)*. 398–411. DOI: https://doi.org/10.1145/2897518.2897568

[15] Sayan Bhattacharya, Monika Henzinger, and Danupon Nanongkai. 2017. 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)*. 470–489. DOI: https://doi.org/10.1137/1.9781611974782.30

[16] Greg Bodwin and Sebastian Krinninger. 2016. Fully dynamic spanners with worst-case update time. In *Proceedings of the European Symposium on Algorithms (ESA)*. 17:1–17:18. DOI: https://doi.org/10.4230/LIPIcs.ESA.2016.17

[17] Keren Censor-Hillel, Elad Haramaty, and Zohar S. Karnin. 2016. Optimal dynamic distributed MIS. In *Proceedings of the Symposium on Principles of Distributed Computing (PODC)*. 217–226. DOI: https://doi.org/10.1145/2933057.2933083

[18] Moses Charikar and Shay Solomon. 2018. Fully dynamic almost-maximal matching: Breaking the polynomial worst-case time barrier. In *Proceedings of the International Colloquium on Automata, Languages and Programming (ICALP)*. 33:1–33:14. DOI: https://doi.org/10.4230/LIPIcs.ICALP.2018.33

[19] Shiri Chechik and Tianyi Zhang. 2019. Fully dynamic maximal independent set in expected poly-log update time. In *Proceedings of the Symposium on Foundations of Computer Science (FOCS)*. 370–381. DOI: https://doi.org/10.1109/FOCS.2019.00031

[20] Benjamin Doerr. 2020. Probabilistic tools for the analysis of randomized optimization heuristics. In *Theory of Evolutionary Computation: Recent Developments in Discrete Optimization*, Benjamin Doerr and Frank Neumann (Eds.). Springer, 1–87. arXiv:1801.06733.

[21] Michael Elkin. 2011. Streaming and fully dynamic centralized algorithms for constructing and maintaining sparse spanners. *ACM Trans. Algor.* 7, 2 (2011), 20:1–20:17. DOI: https://doi.org/10.1145/1921659.1921666

[22] David Eppstein, Zvi Galil, Giuseppe F. Italiano, and Amnon Nissenzweig. 1997. Sparsification—A technique for speeding up dynamic graph algorithms. *J. ACM* 44, 5 (1997), 669–696. DOI: https://doi.org/10.1145/265910.265914

[23] Manuela Fischer. 2017. Improved deterministic distributed matching via rounding. In *Proceedings of the International Symposium on Distributed Computing (DISC)*. 17:1–17:15. DOI: https://doi.org/10.4230/LIPIcs.DISC.2017.17

[24] Sebastian Forster and Gramoz Goranci. 2019. Dynamic low-stretch trees via dynamic low-diameter decompositions. In *Proceedings of the Symposium on Theory of Computing (STOC)*. DOI: https://doi.org/10.1145/3313276.3316381

[25] David Gibb, Bruce M. Kapron, Valerie King, and Nolan Thorn. 2015. Dynamic graph connectivity with improved worst case update time and sublinear space. *CoRR* abs/1509.06464 (2015).

[26] Fabrizio Grandoni, Jochen Könemann, and Alessandro Panconesi. 2008. Distributed weighted vertex cover via maximal matchings. *ACM Trans. Algor.* 5, 1 (2008), 6:1–6:12. DOI: https://doi.org/10.1145/1435375.1435381

[27] Anupam Gupta, Ravishankar Krishnaswamy, Amit Kumar, and Debmalya Panigrahi. 2017. Online and dynamic algorithms for set cover. In *Proceedings of the Symposium on Theory of Computing (STOC)*. 537–550. DOI: https://doi.org/10.1145/3055399.3055493

[28] Manoj Gupta and Richard Peng. 2013. Fully dynamic $(1+\epsilon)$-approximate matchings. In *Proceedings of the Symposium on Foundations of Computer Science (FOCS)*. 548–557. DOI: https://doi.org/10.1109/FOCS.2013.65

[29] Michal Hanckowiak, Michal Karonski, and Alessandro Panconesi. 2001. On the distributed complexity of computing maximal matchings. *SIAM J. Disc. Math.* 15, 1 (2001), 41–57. DOI: https://doi.org/10.1137/S0895480100373121

[30] Monika Henzinger, Sebastian Krinninger, Danupon Nanongkai, and Thatchaphol Saranurak. 2015. Unifying and strengthening hardness for dynamic problems via the online matrix-vector multiplication conjecture. In *Proceedings of the Symposium on Theory of Computing (STOC)*. 21–30. DOI: https://doi.org/10.1145/2746539.2746609

[31] Bruce M. Kapron, Valerie King, and Ben Mountjoy. 2013. Dynamic graph connectivity in polylogarithmic worst case time. In *Proceedings of the Symposium on Discrete Algorithms (SODA)*. 1131–1142. DOI: https://doi.org/10.1137/1.9781611973105.81

[32] Silvio Lattanzi, Benjamin Moseley, Siddharth Suri, and Sergei Vassilvitskii. 2011. Filtering: A method for solving graph problems in MapReduce. In *Proceedings of the Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 85–94. DOI: https://doi.org/10.1145/1989493.1989505

[33] Danupon Nanongkai, Thatchaphol Saranurak, and Christian Wulff-Nilsen. 2017. Dynamic minimum spanning forest with subpolynomial worst-case update time. In *Proceedings of the Symposium on Foundations of Computer Science (FOCS)*. 950–961. DOI: https://doi.org/10.1109/FOCS.2017.92

[34] Ofer Neiman and Shay Solomon. 2016. Simple deterministic algorithms for fully dynamic maximal matching. *ACM Trans. Algor.* 12, 1 (2016), 7:1–7:15. DOI: https://doi.org/10.1145/2700206

[35] Krzysztof Onak and Ronitt Rubinfeld. 2010. Maintaining a large matching and a small vertex cover. In *Proceedings of the Symposium on Theory of Computing (STOC)*. 457–464. DOI: https://doi.org/10.1145/1806689.1806753

[36] Piotr Sankowski. 2004. Dynamic transitive closure via dynamic matrix inverse. In *Proceedings of the Symposium on Foundations of Computer Science (FOCS)*. 509–517. DOI: https://doi.org/10.1109/FOCS.2004.25

[37] Piotr Sankowski. 2007. Faster dynamic matchings and vertex connectivity. In *Proceedings of the Symposium on Discrete Algorithms (SODA)*. 118–126.

[38] Noam Solomon and Shay Solomon. 2021. A generalized matching reconfiguration problem. In *Proceedings of the Innovations in Theoretical Computer Science Conference (ITCS)*. 57:1–57:20. DOI: https://doi.org/10.4230/LIPIcs.ITCS.2021.57

[39] Shay Solomon. 2016. Fully dynamic maximal matching in constant update time. In *Proceedings of the Symposium on Foundations of Computer Science (FOCS)*. 325–334. DOI: https://doi.org/10.1109/FOCS.2016.43

[40] David Wajc. 2020. Rounding dynamic matchings against an adaptive adversary. In *Proceedings of the Symposium on Theory of Computing (STOC)*. 194–207. DOI: https://doi.org/10.1145/3357713.3384258

Received March 2019; revised March 2021; accepted May 2021