

Adaptive Out-Orientations with Applications*

Chandra Chekuri[†] Aleksander Bjørn Christiansen[‡] Jacob Holm[§]
 Ivor van der Hoog[¶] Kent Quanrud[¶] Eva Rotenberg[¶] Chris Schwiegelshohn^{||}

October 2023

Abstract

We give improved algorithms for maintaining edge-orientations of a fully-dynamic graph, such that the maximum out-degree is bounded. On one hand, we show how to orient the edges such that maximum out-degree is proportional to the arboricity α of the graph, in, either, an amortised update time of $\mathcal{O}(\log^2 n \log \alpha)$, or a worst-case update time of $\mathcal{O}(\log^3 n \log \alpha)$. On the other hand, motivated by applications including dynamic maximal matching, we obtain a different trade-off. Namely, the improved update time of either $\mathcal{O}(\log n \log \alpha)$, amortised, or $\mathcal{O}(\log^2 n \log \alpha)$, worst-case, for the problem of maintaining an edge-orientation with at most $\mathcal{O}(\alpha + \log n)$ out-edges per vertex. Finally, all of our algorithms naturally limit the recourse to be polylogarithmic in n and α . Our algorithms adapt to the current arboricity of the graph, and yield improvements over previous work:

Firstly, we obtain deterministic algorithms for maintaining a $(1 + \varepsilon)$ approximation of the maximum subgraph density, ρ , of the dynamic graph. Our algorithms have update times of $\mathcal{O}(\varepsilon^{-6} \log^3 n \log \rho)$ worst-case, and $\mathcal{O}(\varepsilon^{-4} \log^2 n \log \rho)$ amortised, respectively. We may output a subgraph H of the input graph where its density is a $(1 + \varepsilon)$ approximation of the maximum subgraph density in time linear in the size of the subgraph. These algorithms have improved update time compared to the $\mathcal{O}(\varepsilon^{-6} \log^4 n)$ algorithm by Sawlani and Wang from STOC 2020.

Secondly, we obtain an $\mathcal{O}(\varepsilon^{-6} \log^3 n \log \alpha)$ worst-case update time algorithm for maintaining a $(1 + \varepsilon)\text{OPT} + 2$ approximation of the optimal out-orientation of a graph with adaptive arboricity α , improving the $\mathcal{O}(\varepsilon^{-6} \alpha^2 \log^3 n)$ algorithm by Christiansen and Rotenberg from ICALP 2022. This yields the first worst-case polylogarithmic dynamic algorithm for decomposing into $\mathcal{O}(\alpha)$ forests.

Thirdly, we obtain arboricity-adaptive fully-dynamic deterministic algorithms for a variety of problems including maximal matching, $\Delta + 1$ colouring, and matrix vector multiplication. All update times are worst-case $\mathcal{O}(\alpha + \log^2 n \log \alpha)$, where α is the current arboricity of the graph. For the maximal matching problem, the state-of-the-art deterministic algorithms by Kopelowitz, Krauthgamer, Porat, and Solomon from ICALP 2014 runs in time $\mathcal{O}(\alpha^2 + \log^2 n)$, and by Neiman and Solomon from STOC 2013 runs in time $\mathcal{O}(\sqrt{m})$. We give improved running times whenever the arboricity $\alpha \in \omega(\log n \sqrt{\log \log n})$.

Acknowledgements. This research was supported by Independent Research Fund Denmark grant 2020-2023 (9131-00044B) “Dynamic Network Analysis” and the VILLUM Foundation grant (VIL37507) “Efficient Recomputations for Changeful Problems”. This project has additionally received funding from the European Union’s Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 899987. Jacob Holm is part of BARC, supported by the VILLUM Foundation grant 16582. Chris Schwiegelshohn is partially supported by an Independent Research Fund Denmark (DFF) Sapere Aude Research Leader grant No 1051-00106B. Chandra Chekuri is supported by NSF grant CCF-1910149. Kent Quanrud is supported in part by NSF grant CCF-2129816.

*The full version of the paper can be accessed at <https://arxiv.org/abs/2310.18146>

[†]Dept. of Computer Science, University of Illinois, Urbana, IL 61801. chekuri@illinois.edu

[‡]Technical University of Denmark, Kongens Lyngby, Denmark abgch@dtu.dk, vanderhoog@gmail.com, erot@dtu.dk

[§]BARC, University of Copenhagen, Denmark jaho@di.ku.dk

[¶]Dept. of Computer Science, Purdue University, West Lafayette, IN. krq@purdue.edu

^{||}Aarhus University, Aarhus, Denmark. cschwiegelshohn@gmail.com

1 Introduction

In dynamic graphs, one wishes to update a data structure over a graph $G(V, E)$ (or an answer to a specified graph problem) as the graph undergoes local updates such as edge insertions and deletions. One of the fundamental problems is to maintain an orientation of the edges such that the maximum out-degree over all vertices is minimised. While the problem is interesting in its own right, bounded out-degree orientations have a number of applications. First, the problem is closely related to the task of finding the densest subgraph; indeed if the edges can be fractionally oriented, the optimal maximal fractional out-degree is equal to the density $\rho := \frac{|E \cap (S \times S)|}{|S|}$ of the densest subgraph $S \subseteq V$. Secondly, bounded out-degree orientations appear frequently as subroutines for other problems. In particular, there exist a large body of work parameterising the update time of dynamic algorithms for many fundamental problems such as shortest paths [24, 38], vertex cover [43, 47], graph colouring [31, 48], independent set [41], and, most prominently, maximum matching [10, 11, 28, 29, 40, 43, 47] in terms of the arboricity $\alpha := \max_{S \subseteq V, |S| \geq 2} \lceil \frac{|E \cap (S \times S)|}{|S| - 1} \rceil$.

In light of their widespread applicability, maintaining an edge orientation minimising the maximum outdegree is extremely well motivated. In particular, we are interested in algorithms with worst-case deterministic update times, as these can be immediately used as black-box subroutines. In a recent breakthrough result, [45] showed that it is possible to maintain an estimate for the smallest maximum outdegree in $\text{polylog}(n)$ worst case deterministic time by maintaining an estimate for the density of the densest subgraph. Nevertheless, all known results for maintaining an orientation require at least update time $\Omega(\rho) = \Omega(\alpha)$ worst case update time, regardless of whether the algorithm is randomised or not [34]. For dense graphs, this bound may be arbitrarily close to n . Thus, it raises the following question:

QUESTION 1. *Is it possible to maintain an (approximate) minimum out-degree orientation in sublinear deterministic worst case update time?*

1.1 Our Contribution In this paper, we answer the aforementioned question in the affirmative. Specifically, we provide a framework for maintaining approximate out-orientations with various trade-offs between the quality of the out-degree orientation and update time. For the problem of maintaining an out-orientation we obtain:

1. An orientation with maximum out-degree $\mathcal{O}(\alpha)$ in update time $\mathcal{O}(\log^3 n \log \alpha)$.
2. An orientation with maximum out-degree $(1 + \varepsilon)\alpha + 2$ in update time $\mathcal{O}(\varepsilon^{-6} \log^3 n \log \alpha)$.
3. An orientation with maximum out-degree $\mathcal{O}(\alpha + \log n)$ in update time $\mathcal{O}(\log^2 n \log \alpha)$.

The above running times are deterministic and worst-case. Contrary to the previous state-of-the-art result by Sawlani and Wang [45], the recourse of our algorithm, i.e. the number of re-orientations of edges, is polylogarithmic in n (specifically, a $\log n$ factor lower than the running time).

When allowing *amortisation*, we get even better bounds:

4. An orientation with maximum out-degree $\mathcal{O}(\alpha)$ in amortised update time $\mathcal{O}(\log^2 n \log \alpha)$.
5. An orientation with maximum out-degree $\mathcal{O}(\alpha + \log n)$ in amortised update time $\mathcal{O}(\log n \log \alpha)$.

Table 1 gives an overview of our results, and their implications when applied to a selection of algorithmic problems. The latter we briefly discuss in the following.

Densest Subgraph Using the duality between out-degree orientations and maximum subgraph density, we obtain a $(1 + \varepsilon)$ approximate estimate for maximum subgraph density ρ in worst-case update time of $\mathcal{O}(\varepsilon^{-6} \log^3 n \log \rho)$. Additionally, we may output a subgraph H with a density greater than $\rho/(1 + \varepsilon)$ in time linear in the size of H (Lemma 7.1). This recovers (and moderately improves) the recent worst-case algorithm by [45] that has an update time of $\mathcal{O}(\varepsilon^{-6} \log^4 n)$. When allowing amortised analysis, we improve the running time to $\mathcal{O}(\varepsilon^{-4} \log^2 n \log \rho)$ amortised.

Arboricity Decomposition An arboricity decomposition partitions the edge set into a minimum number of forests. The best dynamic algorithms for maintaining an $\mathcal{O}(\alpha)$ arboricity decomposition has an amortised deterministic update time of $\mathcal{O}(\log^2 n)$ due to [31] and an $\mathcal{O}(\sqrt{m} \log n)$ worst case deterministic update time due to [6]. Distinguishing between arboricities 1 and 2 requires $\Omega(\log n)$ time [42, 6]. We substantially improve the worst case update time to $\mathcal{O}(\log^3 n \log \alpha)$.

Dynamic Matrix Vector Multiplication In the Dynamic Matrix Vector Multiplication problem, we are given an $n \times n$ matrix A and an n -vector x . Our goal is to quickly maintain $y = Ax$ in the sense that we can quickly query any entry $y_i = (Ax)_i$, subject to additive updates to the entries of x and A . Interpreting A as the adjacency matrix of a graph with arboricity α , [34] presented an algorithm supporting updates to A in time $\mathcal{O}(\alpha^2 + \log^2 n)$ and updates to x in time $\mathcal{O}(\alpha + \log n)$. We may update A in time $\mathcal{O}(\alpha + \log^2 n \log \alpha)$, improving when $\alpha \in \omega(\log n \sqrt{\log \log n})$.

Maximal Matching A matching is a set of vertex-disjoint edges. A matching M is maximal if no edge of the graph can be added to it without violating the matching property. More so than perhaps any other problem, there exists a large gap between the performance of the state of the art deterministic algorithms vs the state of the art randomised algorithms. Using randomisation, one can achieve a $\mathcal{O}(1)$ amortised [46] and a $\text{polylog}(n)$ worst case update time [9]. Deterministic algorithms so far have only achieved a $\mathcal{O}(\sqrt{m})$ update time for arbitrary graphs [40], or $\mathcal{O}(\alpha^2 + \log^2 n)$ update time where α is the current arboricity of the graph [34]. Because our result explicitly maintains an (approximately) optimal orientation, we improve on known deterministic algorithms whenever $\alpha \in \omega(\log n \sqrt{\log \log n})$ by achieving an update time of $\mathcal{O}(\alpha + \log^2 n \log \alpha)$.

$\Delta + 1$ **Colouring** A fundamental question in many models of computation is how to efficiently compute a $\Delta + 1$ colouring where Δ is the maximum degree of the graph. We present a deterministic algorithm that maintains a $\Delta + 1$ colouring in $\mathcal{O}(\alpha + \log^2 n \log \alpha)$ worst case update time. To the best of our knowledge, this is the first such algorithm that beats the trivial $\mathcal{O}(\Delta)$ update time for uniformly sparse graphs. All other results [13, 32, 12, 48] (discussed in more detail in the appendix) require randomisation, amortisation, or do not yield a $\Delta + 1$ colouring.

Dynamic Problem	Guarantee	Worst case Amortised	Thm.	State-of-the-art comparison
out-orient /density	$\mathcal{O}(\alpha)$	$\mathcal{O}(\log^3 n \log \alpha)$ $\mathcal{O}(\log^2 n \log \alpha)$	Thm. 5.1 Thm. 6.1	$\mathcal{O}(\alpha_{\max})$ in $\mathcal{O}(\alpha_{\max} + \log n)$ [16] $\mathcal{O}(\alpha)$ in $\mathcal{O}(\log^2 n \log \alpha)$ [31] $\mathcal{O}(\alpha)$ in $\mathcal{O}(\log^4 n)$ [45]
out-orient /density	$\mathcal{O}(\alpha + \log n)$	$\mathcal{O}(\log^2 n \log \alpha)$ $\mathcal{O}(\log n \log \alpha)$	Thm. 5.2 Thm. 6.2	$\mathcal{O}(\alpha_{\max} + \log n)$ in $\mathcal{O}(\log n)$ [8] $\mathcal{O}(\alpha + \log n)$ in $\mathcal{O}(\alpha^2 + \log^2 n)$ [34] $\mathcal{O}(\alpha)$ in $\mathcal{O}(\log^4 n)$ [45]
density	$(1 + \varepsilon)\rho$	$\mathcal{O}(\varepsilon^{-6} \log^3 n \log \rho)$ $\mathcal{O}(\varepsilon^{-4} \log^2 n \log \rho)$	Cor. 7.1	$(1 + \varepsilon)\rho$ in $\mathcal{O}(\varepsilon^{-6} \log^4 n)$ [45]
out-orient	$(2 + \varepsilon)\alpha$	$\mathcal{O}(\varepsilon^{-6} \log^3 n \log \alpha)$ $\mathcal{O}(\varepsilon^{-4} \log^2 n \log \alpha)$	Obs. 2	$(2 + \varepsilon)\alpha$ in $\mathcal{O}(\varepsilon^{-6} \log^4 n)$ [45]
out-orient	$(1 + \varepsilon)\alpha + 2$	$\mathcal{O}(\varepsilon^{-6} \log^3 n \log \alpha)$	Thm. 7.1	$\mathcal{O}(\varepsilon^{-6} \alpha^2 \log^3 n)$ [18]
matching	maximal	$\mathcal{O}(\alpha + \log^2 n \log \alpha)$	Cor. 8.1	$\mathcal{O}(\sqrt{m})$ [40] $\mathcal{O}(\alpha^2 + \log^2 n)$ [34]
coloring	$\Delta + 1$ colors	$\mathcal{O}(\alpha + \log^2 n \log \alpha)$	Cor. 8.2	$\mathcal{O}(\Delta)$ (folklore)
arboricity decomp	$\mathcal{O}(\alpha)$	$\mathcal{O}(\log^3 n \log \alpha)$	Cor. 8.5	$\mathcal{O}(\alpha)$ in $\mathcal{O}(\log^2 n)$ [31] $\mathcal{O}(\sqrt{m} \log n)$ [6]
maintain $A \cdot \vec{x}$	–	$\mathcal{O}(\log^2 n \log \alpha)$ upd. A $\mathcal{O}(\alpha + \log n)$ upd. \vec{x} $\mathcal{O}(\alpha + \log n)$ query	Cor. 8.3	$\mathcal{O}(\alpha^2 + \log^2 n)$ updating A $\mathcal{O}(\alpha + \log n)$ query+upd \vec{x} [34]

Table 1: An abbreviated overview of our results, where we compare to state-of-the-art deterministic results. Our results are adaptive to the arboricity α and the maximal subgraph density ρ , and explicitly maintain the out-orientation with low recourse. For out-orientations, note that [45] does not explicitly re-orient edges between updates, but allow for the orientations of edges to be computed upon query time (hence, we marked their result blue).

Related work on dynamic orientations Dynamic out-orientations have been widely studied [16, 29, 37, 18, 17, 8] since they were introduced by Brodal and Fagerberg [16], for maintaining an $\mathcal{O}(\alpha_{\max})$ out-orientation[†] in $\mathcal{O}(\alpha_{\max} + \log n)$ amortised time. Brodal and Berglin [8] improve the time guarantee to worst-case $\mathcal{O}(\alpha_{\max} + \log n)$ time, albeit maintaining an $\mathcal{O}(\alpha_{\max} + \log n)$ out-orientation. The best *adaptive* algorithms, adapting to a changing arboricity, are by Henzinger, Neumann, and Wiese [31] achieving an out-degree of $\mathcal{O}(\alpha)$ and an amortised update time $\mathcal{O}(\log^2 n)$, and by Kopelowitz, Krauthgamer, Porat, and Solomon [34], maintaining an $\mathcal{O}(\alpha + \log n)$ out-orientation with a worst-case update time of $\mathcal{O}(\alpha^2 + \log^2 n)$. Christiansen and Rotenberg [17, 18] lowered the maximum out-degree to $(1 + \varepsilon)\alpha + 2$ incurring a worse update time of $\mathcal{O}(\varepsilon^{-6}\alpha^2 \log^3 n)$.

2 Notation and overview of techniques

Let $G = (V, E)$ be a graph with n vertices and m edges. For any subgraph H of G , we denote by $V(H)$ and $E(H)$ the corresponding vertex and edge set. The *density* of subgraph H is $\rho(H) := \frac{|E(H)|}{|V(H)|}$. The *maximum subgraph density* of G is then the maximum over all H of $\rho(H)$. A closely related measure of uniform sparsity is the *arboricity* of a graph, defined as:

$$\alpha := \max_{H \subseteq G, |V(H)| \geq 2} \left\lceil \frac{|E(H)|}{|V(H)| - 1} \right\rceil$$

A *fractional orientation* \vec{G} of a graph G assigns for every edge (u, v) two weights $d(u \rightarrow v)$ and $d(v \rightarrow u)$ such that $d(u \rightarrow v) + d(v \rightarrow u) = 1$. The out-degree of a vertex u is subsequently defined as $d^+(u) = \sum_{v \in V} d(u \rightarrow v)$. The maximum out-degree of \vec{G} is $\Delta(\vec{G}) := \max_{v \in V} d^+(v)$. Picard & Queyranne [44] show that $\lceil \rho \rceil = \min_{\vec{G}} \Delta(\vec{G})$, and so it follows that $\rho \leq \Delta(\vec{G})$.

An *orientation* is a fractional orientation where $d(u \rightarrow v)$ is either 0 or 1. For brevity, we say that an orientation includes \vec{uv} when $d(u \rightarrow v) = 1$.

For any vertex $u \in V$, we subsequently denote by $N^+(u)$ (resp. $N^-(u)$) all vertices w with $d(u \rightarrow w) = 1$ (resp. $d(w \rightarrow u) = 1$). In an orientation, $d^+(u)$ is the number of edges directed from u (the out-degree). Whenever G is not simple, $d^+(u)$ can be larger than $|N^+(u)|$.

For any integer $b \geq 1$, we denote by G^b the graph G where every edge is duplicated b times. Throughout this paper we maintain for a suitable choice of b , an orientation over G^b . Note that any orientation in G^b induces a fractional orientation on G . We may convert any such fractional orientation in G to an orientation on G by ‘rounding’ every edge (i.e., $d(u \rightarrow v) = 1$ if $d(u \rightarrow v) > d(v \rightarrow u)$, breaking ties arbitrarily). Observe that if the maximum out-degree of an orientation in G^b is some value Δ , then the maximum out-degree of the rounded orientation in G is at most $\Delta / \lceil \frac{b}{2} \rceil \leq 2\Delta/b$. An important theoretical insight for this work are the following linear programs. These dual programs respectively maximise the subgraph density, or minimise the largest fractional out-degree of an edge orientation of G :

Densest Subgraph (DS)

$$\begin{aligned} & \text{maximise} \quad \sum_{\overline{uv} \in E} y_{u,v} && \text{s.t.} \\ & x_u, x_v \geq y_{u,v} && \forall u, v \in V, \overline{uv} \in E \\ & \sum_{v \in V} x_v \leq 1 \\ & x, y \geq 0 \end{aligned}$$

Fractional Orientation (FO)

$$\begin{aligned} & \text{minimise} \quad \rho && \text{s.t.} \\ & d(u \rightarrow v) + d(v \rightarrow u) = 1 && \forall \overline{uv} \in E \\ & \rho \geq d^+(u) = \sum_{v \in V} d(u \rightarrow v) && \forall u \in V \\ & \rho, d(u \rightarrow v), d(v \rightarrow u) \geq 0 \end{aligned}$$

Duality and previous work. The duality between these programs allows for approximating the maximum subgraph density by computing a fractional orientation that aims to minimise ρ . Thus, in an algorithmic sense, we focus on maintaining a fractional orientation of G . This is then achieved by maintaining an integral orientation in a graph with an appropriate number of edge duplicates.

[†]Here α_{\max} denotes the maximum arboricity seen over the whole sequence of operations.

These integral orientations are typically maintained using the following simple, but efficient idea: If one takes a directed path from a high-out-degree vertex to a low-out-degree vertex, then reorienting every edge along this path lowers the out-degree of the high-out-degree vertex while only increasing the out-degree of some vertex of low out-degree. To make this idea constructive, one needs a way to efficiently locate a suitable directed path or *chain* to reorient.

Kopelowitz et al. [34] showed how to locate such chains by maintaining a local condition, namely that $d(u \rightarrow v) > 0$ implies that $d^+(u) \leq d^+(v) + 1$. When the maximum out-degree is small, this local condition can be used to identify short chains. However, when the out-degree becomes large (in dense graphs) this procedure becomes slow. In particular, one can never hope to get a better bound on the chain length than $\Omega(\rho)$. This in turn means that their update times are $\Omega(\rho) = \Omega(n)$. In fact, all of their algorithms have update times that depend on ρ^2 . One ρ stems from the chain lengths and the other from the fact that changes in degrees need to be reported to all out-neighbours in order to efficiently locate the chains.

Sawhani and Wang [45] removed the latter ρ -factor by informing neighbours via a round robin scheme. They then removed the former ρ -factor by instead requiring that $d(u \rightarrow v) > 0$ implies that $d^+(u) \leq d^+(v) + f(\tilde{\rho})$ for some function f and some very precise estimate $\tilde{\rho}$ of the *current* maximum subgraph density. By making the local condition depend on $\tilde{\rho}$, they were able to get chains of much shorter length, namely of length $\mathcal{O}(\varepsilon^{-2} \log n)$. However, for this local condition to yield a small out-degree, one requires that $\tilde{\rho}$ very precisely estimates the current density. To enforce this, Sawhani and Wang [45] maintain $\mathcal{O}(\frac{\log n}{\varepsilon})$ different copies of the graph – each with a different estimate $\tilde{\rho}$. They maintain a pointer to the copy which currently estimates ρ the best.

While this allows Sawhani and Wang [45] to estimate the current maximum subgraph density very well, their approach has several drawbacks. First of all, their algorithm only maintains an *implicit* orientation of the graph in the sense that the algorithm often switches between different copies of the graph each endowed with possibly very different orientations. While this does not matter in the context of density estimation, it matters in the context of using the out-orientation as an algorithmic tool. Firstly, any application run on such an orientation only maintains an implicit representation of the desired outcome, since one continually changes between different copies as updates arrive. Secondly, one has to update the applications across *all* copies meaning that the guarantee on the out-degree is no better than $\mathcal{O}(n)$ in the top-most copy – even if the maximum subgraph density is low. The use of copies also makes the algorithm significantly more complicated.

Our key idea. We show that maintaining a multiplicative local condition, namely that $d^+(u) \leq (1+a) \cdot d^+(v)$ for some chosen value $a < 1$, allows one to get *both* short chains of length $\mathcal{O}(\varepsilon^{-2} \log n \log \rho)$ whilst maintaining a very precise approximation of the maximum subgraph density. Furthermore, this can be achieved completely *explicitly* with low recourse and using only one copy of the graph. This allows us to apply our result to problems that benefit from having an explicit low out-degree orientation such as dynamic maximal matchings and $(\Delta + 1)$ colouring.

Since this multiplicative local condition removes the need for scheduling updates to different copies, the algorithms also become simpler. However, the analysis become significantly more delicate. Sawhani and Wang [45] work with an additive local condition, where the added quantity depends on a very precise estimate $\tilde{\rho}$ of the current density. This allows them, in many places, to essentially reduce the problem-complexity to the case where: a) one has to basically only consider vertices with very large out-degree and b) one can essentially assume that these out-degrees are unchanged, since one is working with a quantity depending on $\tilde{\rho}$. Working with our local condition, however, allows for neither simplification. The local condition is equally “tight” for every vertex, and it is very sensitive to changes in degrees at both endpoints of an edge. This means that one has to be very careful, when analysing the algorithms – especially when the vertices have low degree.

Multiplicative local conditions We consider two local conditions that we want to maintain for an integral orientation. The first has both an additive and multiplicative term. The second has only a multiplicative term. In the first case, we require that $d(u \rightarrow v) > 0$ implies that $d^+(u) \leq (1+a) \cdot d^+(v) + c$. The benefit of the additive term is that for any new edge (u, v) , we may always orient the edge towards either u or v without violating this local condition between u and v . The downside of this approach is that it leads to less accurate estimations of α and ρ . In the second case, we require that $d(u \rightarrow v) > 0$ implies that $d^+(u) \leq (1+a) \cdot d^+(v)$. If both $d^+(u)$ and $d^+(v)$ are small, it may be that $d^+(u) + 1 > (1+a) \cdot d^+(v)$ and $d^+(v) + 1 > (1+a) \cdot d^+(u)$. Thus, when adding an edge (u, v) we cannot orient the edge without violating our local condition. This significantly complicates the analysis.

Indeed, this complication means that we can only guarantee the multiplicative condition holds *between* updates

to G , and is not maintained as an invariant as we perform updates to G^b . Hence, to get a simple recursive algorithm to work, we have to instead work with a thresholded local condition, where we allow edges between vertices of small enough degree to get a direction in order to handle the above problem. We show that maintaining such a thresholded local condition is actually equivalent to maintaining the multiplicative condition between updates to G . However, working with this thresholded condition requires one to be careful. To illustrate this, we briefly sketch how the algorithms work: suppose first that every vertex has perfect information about the degrees of all other vertices. Then, it can immediately identify if incrementing/decrementing its degree causes the local condition to be violated. If so, it can then reorient a violated edge, thus restoring its degree. This solves the problem for this vertex, but might move the problem to some other vertex. The key property however is that this vertex has a (significantly) smaller degree in the incremental case, or (significantly) higher degree in the decremental case. Hence, this cascade cannot continue many times, thus yielding a short chain. Every vertex, however, does not necessarily have access to the degree of all of its neighbours. Thus the algorithm has to supply this information somehow. We do so in 3 different ways: by naively informing and checking all out-neighbours of degree changes (reminiscent of the approach of Kopelowitz et al. [35]), by updating and checking estimates lazily in a round robin fashion (similar to Sawlani and Wang [45]) and in an amortised fashion by only checking every time a degree has changed substantially. The two last schemes demand that we at all times work with degree estimates that are not precise. Doing so is quite straightforward with a purely additive local condition, due to the simplifications mentioned earlier, but it is significantly more involved in the multiplicative case: here the conditions are very sensitive to degree changes, and so to make the analysis work, we have to be very precise about at what time a certain condition on the degree holds. Particularly so, when the degrees are small. This is further complicated by the fact that we now work with a thresholded local condition and thusly have to ensure that our analysis can handle all paradigms of the condition.

In Section [3] we analyse the effect of maintaining an integral orientation that satisfies our local condition (that for all (u, v) , $d(u \rightarrow v) > 0$ implies $d^+(u) \leq (1 + a) \cdot d^+(v) + c$). We present a general theorem showing the impact of our local condition, parametrized by a and c . Let Δ be the maximal out-degree in our graph. We immediately apply this theorem to show that for $a^{-1} \in \mathcal{O}(\log n)$: $\Delta \in \mathcal{O}(\rho + \log n)$ (for $c \in \mathcal{O}(1)$) or $\Delta \in \mathcal{O}(\rho)$ (for $c = 0$). In Section [7] we show that choosing $c = 0$ and $a^{-1} \in \mathcal{O}(\varepsilon^{-2} \log n)$ allows us to maintain a fractional orientation of the graph where the maximal out-degree $\Delta \leq (1 + \varepsilon)\rho$. By naively rounding the fractional out-degrees, this implies that we can maintain an integral orientation where the out-degree $\Delta \leq (2 + \varepsilon)\rho$.

2.1 Parameterisation of the Algorithm We now introduce several components of the algorithm and analysis that can be specified to obtain various trade-offs between quality of the out-orientation and update time. Our algorithms have the two main parameters: η and a positive integer b . We maintain a graph G^b and an orientation \vec{G}^b where one of the following invariants holds:

INVARIANT 0. We maintain an orientation \vec{G}^b where for every directed edge \vec{uv} in \vec{G}^b :

$$d^+(u) \leq (1 + \eta \cdot b^{-1}) \cdot d^+(v).$$

INVARIANT 1. We maintain an orientation \vec{G}^b where for every directed edge \vec{uv} in \vec{G}^b :

$$d^+(u) \leq (1 + \eta \cdot b^{-1}) \cdot d^+(v) + 2.$$

Throughout the paper, we denote $\theta = 0$ if we are maintaining Invariant [0] and $\theta = 1$ otherwise. This way, we maintain Invariant θ by maintaining $d^+(u) \leq (1 + \eta \cdot b^{-1}) \cdot d^+(v) + 2\theta$. The tighter the inequalities are, the closer the maximum out-degree of the maintained out-orientation is to the maximum subgraph density. Hence, setting $\theta = 0$ will give a better approximation than $\theta = 1$.

Note that regardless of the choice of parameters, not all graphs have an orientation that satisfies Invariant [0]. E.g. any orientation of the graph consisting of a single edge has a directed edge \vec{uv} with $1 = d^+(u) > (1 + \eta \cdot b^{-1}) \cdot d^+(v) = 0$. For convenience we will therefore need the following slightly relaxed invariant, which we show in Section [4] is satisfiable for $\theta = 0$ (and therefore for all $\theta \geq 0$) as long as $0 < \frac{b}{\eta} \leq \lfloor \frac{b}{2} \rfloor$, i.e. when $b \geq 2$ is even and $\eta \geq 2$ or when $b \geq 3$ and $\eta \geq 3$ (or more precisely $\eta \geq \frac{2b}{b-1}$).

INVARIANT θ' . We maintain an orientation \vec{G}^b where for every directed edge \vec{uv} in \vec{G}^b :

$$d^+(u) \leq \max\left\{(1 + \eta \cdot b^{-1}) \cdot d^+(v) + 2\theta, \left\lfloor \frac{b}{2} \right\rfloor\right\}.$$

The point is that for each update to G this lets us do updates to \vec{G}^b one edge at a time, all the while satisfying Invariant θ' and when we are done the resulting graph satisfies Invariant θ because of the following Lemma.

LEMMA 2.1. Let G^b be a graph that can be obtained from a graph G by replacing each edge with b copies. For all $\theta \geq 0$, any orientation \vec{G}^b of G^b that satisfies Invariant θ' also satisfies Invariant θ .

Proof. Suppose \vec{G}^b satisfies Invariant θ' and let \vec{uv} be any edge in \vec{G}^b . If $d^+(v) < \lceil \frac{b}{2} \rceil$ then (because each edge in G^b is duplicated b times) $d^+(u) \geq b - d^+(v) > \lfloor \frac{b}{2} \rfloor$. So by Invariant θ' we have $d^+(u) \leq (1 + \eta \cdot b^{-1}) \cdot d^+(v) + 2\theta$ and Invariant θ is satisfied for \vec{uv} . Otherwise $d^+(v) \geq \lceil \frac{b}{2} \rceil$ and by Invariant θ' $d^+(u) \leq \max\{(1 + \eta \cdot b^{-1}) \cdot d^+(v) + 2\theta, \lfloor \frac{b}{2} \rfloor\} = (1 + \eta \cdot b^{-1}) \cdot d^+(v) + 2\theta$ thus Invariant θ is satisfied for \vec{uv} . \square

3 A Structural Theorem

In this section, we formally establish the relationship between maintaining Invariant θ for a graph \vec{G}^b , and the corresponding estimate of the density and arboricity of the graph G . The following theorem is our result in its most general form: allowing for $(1 + \varepsilon)$ -approximations of ρ and more. Skip ahead to Corollaries 3.1+3.2 for a comprehensible application of the variables.

THEOREM 3.1. Let G be a graph and let G^b be G with each edge duplicated b times. Let ρ_b be the maximum subgraph density of G^b . Let \vec{G}^b be any orientation of G^b which has the following invariant: for some $c \geq 0$, every directed edge \vec{uv} satisfies $d^+(u) \leq (1 + \eta \cdot b^{-1}) \cdot d^+(v) + c$.

Then for any $\gamma > 0$ there exists a value $k_{\max} \leq \log_{1+\gamma} n$ for which:

$$(1 + \eta \cdot b^{-1})^{-k_{\max}} \Delta(\vec{G}^b) \leq (1 + \gamma)\rho_b + c(\eta^{-1} \cdot b + 1).$$

Proof. Let $G^b = (V, E^b)$. We define for non-negative integers i the sets:

$$T_i := \left\{ v \in V \mid d^+(v) \geq \Delta(\vec{G}^b) \cdot (1 + \eta \cdot b^{-1})^{-i} - c \sum_{j=1}^i (1 + \eta \cdot b^{-1})^{-j} \right\}$$

Observe that for all non-negative integers $0 \leq i < j$, $T_i \subseteq T_j$. Moreover, observe that T_0 contains at least one element (the element of \vec{G}^b with maximum out-degree), and each T_i at most n elements (since they can contain at most all vertices of G). Let k be the smallest integer such that $|T_{k+1}| < (1 + \gamma)|T_k|$. It follows that k is upper bounded by the value $k_{\max} = \log_{(1+\gamma)} n$.

In order to bound the maximum out-degree of \vec{G}^b , we want to show that no edges can be oriented from T_k to a vertex not in T_{k+1} . To do so, we assume two such candidates $u \in T_k$ and $v \notin T_{k+1}$, and show that \vec{uv} violates: $d^+(u) \leq (1 + \eta \cdot b^{-1}) \cdot d^+(v) + c$. Per assumption we have

$$d^+(u) \geq (1 + \eta \cdot b^{-1})^{-k} \Delta(\vec{G}^b) - c \sum_{j=1}^k (1 + \eta \cdot b^{-1})^{-j} \quad (\text{Since } u \in T_k)$$

and

$$d^+(v) < (1 + \eta \cdot b^{-1})^{-1} (1 + \eta \cdot b^{-1})^{-k} \Delta(\vec{G}^b) - c \sum_{j=1}^{k+1} (1 + \eta \cdot b^{-1})^{-j}. \quad (\text{Since } v \notin T_{k+1})$$

It follows that

$$\begin{aligned} (1 + \eta \cdot b^{-1})d^+(v) + c &< (1 + \eta \cdot b^{-1})^{-k} \Delta(\vec{G}^b) - c \sum_{j=0}^k (1 + \eta \cdot b^{-1})^{-j} + c \\ &= (1 + \eta \cdot b^{-1})^{-k} \Delta(\vec{G}^b) - c \sum_{j=1}^k (1 + \eta \cdot b^{-1})^{-j} \leq d^+(u). \end{aligned}$$

This would violate the assumed invariant of \vec{G}^b . Hence for any $u \in T_k$ and any edge \vec{uv} , we have $v \in T_{k+1}$ and thus: $\sum_{u \in T_k} d^+(u) \leq |E^b[T_{k+1}]|$. Finally, we can bound the density ρ_b as:

$$\begin{aligned} \rho_b &= \max_{\emptyset \subset S \subseteq V} \frac{|E^b[S]|}{|S|} \geq \frac{|E^b[T_{k+1}]|}{|T_{k+1}|} \geq \frac{\sum_{u \in T_k} d^+(u)}{(1 + \gamma)|T_k|} \\ &\geq \frac{|T_k| \cdot \left((1 + \eta \cdot b^{-1})^{-k} \Delta(\vec{G}^b) - c \sum_{j=1}^k (1 + \eta \cdot b^{-1})^{-j} \right)}{(1 + \gamma)|T_k|}. \end{aligned}$$

We find $(1 + \gamma)\rho_b + \frac{c}{1 - \frac{1}{1 + \eta \cdot b^{-1}}} \geq (1 + \eta \cdot b^{-1})^{-k_{\max}} \Delta(\vec{G}^b)$, which concludes the proof. \square

The parameter γ is needed to get a $(1 + \varepsilon)$ -approximation later on, where we will require that $\gamma = \Theta(\varepsilon)$. For now, one can just think of γ as being a constant. In fact in the following corollaries, we will choose γ so $1 + \gamma = e$.

COROLLARY 3.1. *Denote by ρ the density of G . For any η and b such that $\eta b^{-1} \in \mathcal{O}(\frac{1}{\log n})$, we have that Invariant $\boxed{\theta}$ for the graph G^b implies: $\Delta(\vec{G}^b) \in \mathcal{O}(b\rho)$ and $\Delta(\vec{G}) \in \mathcal{O}(\rho)$.*

Proof. Set $\gamma = e - 1$, let $k_{\max} \leq \log_{(1+\gamma)} n = \log_e n$ be as in Theorem $\boxed{3.1}$. By our choice of η and b there exists a constant $s > 0$ such that $\eta b^{-1} \leq \frac{s}{\log_e n}$ for all $n \geq 1$, thus by Theorem $\boxed{3.1}$ (with $c = 0$) we now have

$$\Delta(\vec{G}^b) \leq (1 + \eta \cdot b^{-1})^{k_{\max}} (1 + \gamma)\rho_b \leq e^{\eta \cdot b^{-1} \cdot k_{\max}} (1 + \gamma)\rho_b \leq e^{s+1} \rho_b \in \mathcal{O}(\rho_b).$$

Finally, we note that per definition of subgraph density, $\rho_b = b \cdot \rho$. For all \vec{uv} in \vec{G} , there must be at least $\lceil \frac{b}{2} \rceil$ edges in \vec{G}^b from u to v (else, \vec{G} would include the edge \vec{vu} instead). It immediately follows that the out-degree of u in \vec{G} is at most $d^+(u) \cdot \lceil \frac{b}{2} \rceil^{-1} \in \mathcal{O}(b^{-1} \cdot \rho_b) = \mathcal{O}(\rho)$. \square

COROLLARY 3.2. *Denote by ρ the density of G . Let $b = 1$ and $\eta = \frac{1}{\log_e(n)}$.*

Whenever Invariant $\boxed{1}$ holds for the graph $\vec{G} = \vec{G}^b$, it must be that: $\Delta(\vec{G}) \in \mathcal{O}(\rho + \log n)$.

Proof. Set $\gamma = e - 1$, let k_{\max} be as in Theorem $\boxed{3.1}$. By our choice of η and b , $\eta \cdot b^{-1} = \frac{1}{\log_e n} = \frac{1}{\log_{(1+\gamma)} n}$. Thus by Theorem $\boxed{3.1}$ (with $c = 2$) we now have

$$\begin{aligned} \Delta(\vec{G}) &= \Delta(\vec{G}^b) \leq (1 + \eta \cdot b^{-1})^{k_{\max}} ((1 + \gamma)\rho_b + c(\eta^{-1} \cdot b + 1)) \\ &\leq e^{\eta \cdot b^{-1} \cdot k_{\max}} ((1 + \gamma)\rho_b + c(\eta^{-1} \cdot b + 1)) \\ &\leq e(e \cdot \rho_b + c(\eta^{-1} \cdot b + 1)) \\ &\in \mathcal{O}(\rho + \log n) \end{aligned}$$

\square

4 A Simple Algorithm for Maintaining the Invariants

We first provide a simple worst-case $\mathcal{O}(\rho \log \rho \cdot \text{polylog}(n))$ algorithm (where ρ is the maximum subgraph density) to maintain Invariant θ in \vec{G}^b (i.e. we maintain one chosen invariant). Our data structure is purposefully more

complicated than necessary here, to illustrate its use in future sections. Subsequent sections slightly adjust the algorithms. Crucially, the bound on the recursive depth of our functions applies throughout the paper. Recall that G^b is the graph G with edges duplicated b times. For convenience, we set $\lambda = \eta b^{-1}/64$ in the rest of the paper, and note that $(1 + \lambda)^5 \leq 1 + \eta b^{-1} \leq 2$. We maintain Invariant θ using a data structure storing for all vertices u :

- (a) The value $d^+(u)$ of the current orientation \vec{G}^b ,
- (b) The set $N^+(u)$ in arbitrary order, and
- (c) The set $N^-(u)$ in a sorted doubly linked list of buckets $B_j(u)$. Each bucket $B_j(u)$ contains, as a doubly linked list in arbitrary order, all $w \in N^-(u)$ where $j = \lfloor \log_{(1+\lambda)} d^+(u) \rfloor$. The vertex u has a pointer to the bucket $B_i(u)$ with $i = \lfloor \log_{(1+\lambda)} \max\{(1 + \lambda)d^+(u), \lfloor \frac{b}{4} \rfloor\} \rfloor$.

We run Algorithms [1](#)+[2](#) on the graph G . These invoke Algorithms [3](#)+[4](#) which in turn add directed edges to and remove them from G^b (Algorithms [5](#)+[6](#)). In our recursive algorithm calls, we may assume that for any edge insertion (u, v) in G^b , we call $\text{Insert}(\vec{uv})$ whenever $d^+(u) \leq d^+(v)$. Recall that θ , η and b are parameters that are set beforehand:

Algorithm 1: $\text{Insert}(\text{edge } (u, v) \text{ in } G)$ <hr/> for $i \in [b]$ do if $d^+(u) \leq d^+(v)$ $\text{Insert}(\vec{uv})$ else $\text{Insert}(\vec{vu})$	Algorithm 2: $\text{Delete}(\text{edge } (u, v) \text{ in } G)$ <hr/> for $i \in [b]$ do if $u \in N^-(v)$ $\text{Delete}(\vec{vu})$ else $\text{Delete}(\vec{uv})$
Algorithm 3: $\text{Insert}(\vec{uv})$, where $d^+(u) \leq d^+(v)$ <hr/> $\text{Add}(\vec{uv})$ $x \leftarrow \arg \min\{d^+(w) \mid w \in N^+(u)\}$ if $d^+(u) > \max\{(1 + \lambda) \cdot d^+(x) + \theta, \lfloor \frac{b}{4} \rfloor\}$ $\text{Remove}(\vec{ux})$ {restores $d^+(u)$ } $\text{Insert}(\vec{xu})$ else for all $w \in N^+(u)$ do Update $d^+(u)$ in $\text{Buckets}(N^-(w))$	Algorithm 4: $\text{Delete}(\vec{uv})$ <hr/> $\text{Remove}(\vec{uv})$ $x \leftarrow \text{First}(\text{Max}(\text{Buckets}(N^-(u))))$ if $d^+(x) > \max\{(1 + \lambda) \cdot d^+(u) + \theta, \lfloor \frac{b}{4} \rfloor\}$ $\text{Add}(\vec{ux})$ {restores $d^+(u)$ } $\text{Delete}(\vec{xu})$ else for all $w \in N^+(u)$ do Update $d^+(u)$ in $\text{Buckets}(N^-(w))$
Algorithm 5: $\text{Add}(\vec{uv})$ <hr/> $d^+(u) = d^+(u) + 1$ if $\nexists \vec{uv} \in \vec{G}^b$ Add u to $N^-(v)$ and v to $N^+(u)$ Add one edge \vec{uv} to \vec{G}^b .	Algorithm 6: $\text{Remove}(\vec{uv})$ <hr/> $d^+(u) = d^+(u) - 1$ Remove one edge \vec{uv} from \vec{G}^b . if $\nexists \vec{uv} \in \vec{G}^b$ Remove u from $N^-(v)$, v from $N^+(u)$

DEFINITION 1. We count time in discrete steps. A new time step starts just before Algorithm [1](#) calls Insert or Algorithm [2](#) calls Delete . For a time t , we denote for any variable ϕ in our code by ϕ_t its value before the invoking insertions (or deletions) at time t . E.g., for a vertex w , $d^+(w)_t$ is the out-degree before invoking insertions at time t , and $d^+(w)_{t+1}$ is the out-degree just after.

4.1 Maintaining Invariant [1](#). We show that by setting $\theta = 1$ (and choosing η and b carefully) we maintain Invariant [1](#):

THEOREM 4.1. Let G be a dynamic graph and ρ be the density of G at time t . We can choose our variables $\theta = 1$, $b = 1$ and $\eta \in \Theta(\log n)$ to maintain an out-orientation $\vec{G}^b = \vec{G}$ in $O((\rho + \log n) \cdot \log n \cdot \log \rho)$ time per update in G such that Invariant [1](#) holds for \vec{G} . Moreover:

- $\forall u$, the out-degree $d^+(u)_{t+1}$ in \vec{G} is at most $\mathcal{O}(\rho + \log n)$, (i.e. $\Delta(\vec{G}) \in \mathcal{O}(\rho + \log n)$)

Proof. Invariant [1](#) demands that $\forall \vec{u}\vec{v}$ we maintain $d^+(u) < (1 + \eta b^{-1})d^+(v) + 2$. Corollary [3.2](#) implies that if after time t we maintain Invariant [1](#), then we obtain the desired upper bound on all $d^+(u)_{t+1}$. What remains is to show that our algorithms maintain Invariant [1](#) in the desired runtime. We do this in three steps as we show:

Correctness: Our algorithms maintain Invariant [1](#) in G^b ,

Recursive depth: Algorithms [3](#)+[4](#) have a recursive depth of $\mathcal{O}(\lambda^{-1} \cdot \log \rho)$, and

Time: Algorithms [3](#)+[4](#) spend $\mathcal{O}(\rho + \log n)$ time before entering the recursion.

We prove these three properties for deletions only. Invoking $\text{Delete}(\overrightarrow{x_0 x_i})$ may cause us to recursively invoke $\text{Delete}(\overrightarrow{x_{i+1} x_i})$: flipping a backward chain in G^b from x_0 . Only the final vertex x_f in this chain decreases its out-degree once we terminate. For insertions we flip a forward chain $\overrightarrow{x_i x_{i+1}}$, which is handled symmetrically.

Correctness. We show that we maintain Invariant [1](#). Suppose that we terminate at a vertex x_f . Then after our sequence of flips, the vertex x_f is the only vertex that changed its out-degree (i.e. only for x_f : $d^+(x_f)_{t+1} = d^+(x_f)_t - 1$). Because our algorithm terminated and $b = 1$, for $x \leftarrow \text{First}(\text{Max}(\text{Buckets}(N^-(x_f))))$, $d^+(x)_t \leq \max\{(1 + \lambda)(d^+(x_f)_t - 1) + \theta, \lfloor \frac{b}{4} \rfloor\}$. For all $w \in N^-(x_f)$: $d^+(w)_{t+1} \leq (1 + \lambda)d^+(x)_{t+1}$. It follows $d^+(w)_{t+1} \leq \max\{(1 + \lambda)^2 d^+(x_f)_{t+1} + 2, \lfloor \frac{b}{2} \rfloor\}$. We may apply Lemma [2.1](#) to conclude that, once terminated, we satisfy Invariant [1](#).

Recursive depth. What remains is to upper bound the recursive depth of our algorithm, proving termination. Our code implies that for all i : $d^+(x_{i+1})_t > (1 + \lambda)(d^+(x_i)_t - 1) + \theta$. Thus $d^+(x_{i+1})_t \geq d^+(x_i)_t + 1$. Let x_s be the last vertex in the chain where $d^+(s)_t \in \mathcal{O}(\log n)$. The fact that out-degrees are integer and strictly increasing along the backward chain, implies that there are $\mathcal{O}(\log n)$ vertices preceding s . If $f = s$, the recursive depth is $\mathcal{O}(\log n) = \mathcal{O}(\lambda^{-1})$ per definition.

Otherwise, we note that before this sequence of updates, we satisfied Invariant [1](#) and thus (by Corollary [3.2](#)) know that for all i : $d^+(x_i)_t \in \mathcal{O}(\rho + \log n)$. If there exist vertices x_i with $i > s$, then $\rho \in \Omega(\log n)$ and thus $\mathcal{O}(\rho + \log n) = \mathcal{O}(\rho)$. Now we consider all $i > s$. We know that $d^+(x_{i+1})_t > (1 + \lambda)(d^+(x_i)_t - 1)$. Thus, (using $d^+(x_i)_t \geq d^+(x_{i-1})_t + 1$) we get that: $d^+(x_{i+1})_t > (1 + \lambda)d^+(x_{i-1})_t$. It follows that there are at most $\log_{(1+\lambda)} \mathcal{O}(\rho) = \mathcal{O}(\lambda^{-1} \log \rho)$ vertices in the chain of flipped edges: which upper bounds our recursive depth.

Time spent. Whenever we insert a vertex $v \in N^-(u)$, it is either because we added the edge (u, v) to G (occurring once) or, because we flipped an edge $\vec{u}\vec{v}$. In the first case, we may afford spending $\mathcal{O}(\log_{(1+\lambda)} d^+(v)) = \mathcal{O}(\lambda^{-1} \log(b\rho + \log n)) = \mathcal{O}(\lambda^{-1} \log n)$ time searching through all buckets for the bucket containing v . In the latter case, for Insert we have $\max\{(1 + \lambda)d^+(u)_t, \lfloor \frac{b}{4} \rfloor\} < d^+(v)_t + 1 \leq \max\{(1 + \lambda)d^+(u)_t, \lfloor \frac{b}{4} \rfloor\} + 1$, and for Delete we have $\max\{(1 + \lambda)(d^+(u)_t - 1), \lfloor \frac{b}{4} \rfloor\} < d^+(v)_t < (1 + \lambda) \max\{(1 + \lambda)(d^+(u)_t - 1), \lfloor \frac{b}{4} \rfloor\}$. Using the pointer from u to the bucket $B_i(u)$ where $i = \lfloor \log_{(1+\lambda)} \max\{(1 + \lambda)d^+(u), \lfloor \frac{b}{4} \rfloor\} \rfloor$, we may insert v into the correct bucket in $\mathcal{O}(1)$ time. For each call of $\text{Delete}(\overrightarrow{x_{i+1} x_i})$, we spend $\mathcal{O}(1)$ time retrieving the vertex x before we recurse. For the vertex x_f at the end of the recursion, we consider all $\mathcal{O}(\rho + \log n)$ vertices $w \in N^+(x_f)$. We update the bucket that x_f is in. Denote by $r(x_f)_{t+1} = \lfloor \log_{(1+\lambda)} d^+(x_f)_t \rfloor$ the rank of f (i.e., the index of each bucket contained x_f at time t). The rank of x_f changes by at most 1, hence we may update our data structure in $\mathcal{O}(d^+(f)_{t+1})$ time.

For each call of $\text{Insert}(\overrightarrow{x_{i+1} x_i})$, we spend $\mathcal{O}(d^+(x_{i+1})_t) = \mathcal{O}(\rho + \log n)$ time retrieving the vertex x_{i+2} before we recurse. Updating the data structure again takes $\mathcal{O}(1)$ time per updated element. It follows that the total time spent adding or removing an arc in G^b is $\mathcal{O}((\rho + \log n) \cdot \lambda^{-1} \log \rho) = \mathcal{O}((\rho + \log n) \log n \log \rho)$. Since $b = 1$, the theorem follows. \square

4.2 Maintaining Invariant [0](#). We show that by setting $\theta = 0$ (and choosing η and b carefully) we maintain Invariant [0](#):

THEOREM 4.2. *Let G be a dynamic graph and ρ be the density of G at time t . We can choose our variables $\theta = 0$, $\eta = 3$, and $b \in \Theta(\log n)$, $b \geq 2$ to maintain an out-orientation \vec{G}^b in $\mathcal{O}(b \cdot \rho \cdot \lambda^{-1} \cdot \log \rho) = \mathcal{O}(\rho \cdot \log^2 n \log \rho)$ time per update in G , maintaining Invariant [0](#) for \vec{G}^b with:*

- $\forall v$, the out-degree $d^+(v)_{t+1}$ in \vec{G}^b is at most $\mathcal{O}(b \cdot \rho)$, and
- $\forall u$, the out-degree of u in \vec{G} is at most $\mathcal{O}(\rho)$.

Proof. We show that at all times we maintain Invariant [0](#)' for $\theta = 0$. Corollary [3.1](#) and our choice of variables, implies the desired upper bound on the out-degree of each vertex. We again consider:

Correctness: Our algorithms maintain Invariant θ' in G^b ,

Recursive depth: Algorithms $\text{\textcolor{red}{3}}+\text{\textcolor{red}{4}}$ have a recursive depth of $\mathcal{O}(\lambda^{-1} \cdot \log \rho)$, and

Time: Algorithms $\text{\textcolor{red}{3}}+\text{\textcolor{red}{4}}$ spend $\mathcal{O}(\rho)$ time before entering the recursion.

We show the proof for deletions. Again, the proof for insertions is symmetrical (flipping a forward chain). Invoking $\text{Delete}(\overrightarrow{x_0 v})$ may cause us to recursively invoke $\text{Delete}(\overrightarrow{x_{i+1} x_i})$: flipping a backward chain in G^b from x_0 . Only the last vertex x_f in this chain decreases its out-degree once we terminate.

Correctness. Suppose that we terminate at a vertex x_f . Then after our sequence of flips, only the vertex x_f changed its out-degree (i.e. only for x_f : $d^+(x_f)_{t+1} = d^+(x_f)_t - 1$). Because our algorithm terminated, for $x \leftarrow \text{First}(\text{Max}(\text{Buckets}(N^-(x_f))))$ it must be that: $d^+(x)_t \leq \max\{(1 + \lambda)(d^+(x_f)_t - 1) + \theta, \lfloor \frac{b}{4} \rfloor\}$. It follows that for all vertices $w \in N^-(x_f)$: $d^+(w)_t \leq (1 + \lambda) \max\{(1 + \lambda)(d^+(x_f)_t - 1) + \theta, \lfloor \frac{b}{4} \rfloor\}$. Substituting $d^+(x_f)_t$ for $d^+(x_f)_{t+1}$ and using that by our choice of parameters, $(1 + \lambda)^2 \leq 1 + \eta b^{-1} \leq 2$ now gives that for all $w \in N^-(x_f)$: $d^+(w)_t \leq \max\{(1 + \eta b^{-1})d^+(x_f)_{t+1} + 2\theta, \lfloor \frac{b}{2} \rfloor\}$. By Lemma $\text{\textcolor{red}{2.1}}$ this implies that we maintain Invariant $\text{\textcolor{red}{0}}$.

Recursive depth. What remains is to upper bound the recursive depth of our algorithm. Let x_s be the first vertex in the chain where $d^+(x_s)_t \geq \lfloor \frac{b}{4} \rfloor + 2$. Note that per definition of our algorithm, $d^+(x_1)_t \geq \lfloor \frac{b}{4} \rfloor + 1$. Thus, for all $i \geq 1$: $d^+(x_{i+1})_t \geq d^+(x_i)_t + 1$ and $s \leq 2$. We now make a case distinction. If $f \in \mathcal{O}(\lambda^{-1})$ then per definition, the recursive depth is $\mathcal{O}(\lambda^{-1})$.

Otherwise, for all $i > 2$ it must be that $d^+(x_{i+1})_t > (1 + \lambda)(d^+(x_i) - 1) \geq (1 + \lambda)d^+(x_{i-1})$. Before this sequence of updates, we satisfied Invariant $\text{\textcolor{red}{0}}$. Thus, by Corollary $\text{\textcolor{red}{3.1}}$ for all i , $d^+(x_i)_t \in \mathcal{O}(\rho)$, and thus the recursive depth is $\mathcal{O}(\log_{1+\lambda} d^+(x_1)_t) = \mathcal{O}(\lambda^{-1} \log \rho)$.

Time spent. The proof upper bounding the time spent is identical to that of Theorem $\text{\textcolor{red}{4.2}}$. The one exception being, that the out-degree $d^+(u)$ for all vertices u is at most ρ . Thus, the running time per update in G^b is $\mathcal{O}(\rho \cdot \lambda^{-1} \log \rho) = \mathcal{O}(\rho \log n \log \rho)$. Each update in G triggers $\Theta(\log n)$ updates in G^b and so the runtime follows. \square

5 Improved worst case algorithms

We adapt the algorithm of Section $\text{\textcolor{red}{4}}$, replacing the algorithms for inserting and deleting directed edges in G^b to update our running time. We store $u \in N^-(v)$ in buckets determined not by the actual out-degrees $d^+(u)$ but rather by an approximation of what we call the *out-rank* $r(u) = \lfloor \log_{(1+\lambda)} d^+(u) \rfloor$.

DEFINITION 2. For each vertex v , for all vertices $u \in N^-(v)$, we define the perceived out-rank $r_v(u)$ as some integer stored in v for $u \in N^-(v)$ (which we show is at most 1 removed from $r(u)$).

In this section, We maintain for all u :

- (a) The *exact* value $d^+(u)$ of the current orientation \overrightarrow{G}^b ,
- (b) The set $N^+(u)$ in a linked list and a pointer some current ‘position’ in the linked list.
- (c) The set $N^-(u)$ in a doubly linked list of buckets $B_j(u)$ sorted by j from high to low. Each bucket $B_j(u)$ contains, as a doubly linked list in arbitrary order, all $w \in N^-(u)$ where $r_u(w) = j$. The vertex u has a pointer to the bucket $B_i(u)$ with $i = \lfloor \log_{(1+\lambda)} \max\{(1 + \lambda)d^+(u), \lfloor \frac{b}{4} \rfloor\} \rfloor$.

Any update in G invokes Algorithms $\text{\textcolor{red}{1}}+\text{\textcolor{red}{2}}$. These algorithms now invoke Algorithm $\text{\textcolor{red}{7}}$ or $\text{\textcolor{red}{8}}$ (instead of $\text{\textcolor{red}{3}}$ or $\text{\textcolor{red}{4}}$). These two in turn invoke the normal add and remove functions (Algorithm $\text{\textcolor{red}{5}}+\text{\textcolor{red}{6}}$). Whenever we add a vertex w to a set $N^-(u)$, we set $r_u(w) = r(w)$. And when we add a vertex v to a set $N^+(u)$, we do so in the position immediately *before* the current position, so it becomes the last one we visit when we round-robin over $N^+(u)$.

Algorithm 7: Insert(\vec{uv})	Algorithm 8: Delete(\vec{uv})
Add(\vec{uv}) for x in next $\lceil \frac{2}{\lambda} \rceil$ neighbours in $N^+(u)$ do if $d^+(u) > \max\{(1 + \lambda) \cdot d^+(x) + \theta, \lfloor \frac{b}{4} \rfloor\}$ Remove(\vec{ux}) Insert(\vec{xu}) BREAK for all x visited in the previous loop do $r_x(u) = r(u)$ Move u to bucket $B_{r_x(u)}$ in $N^-(x)$	Remove(\vec{uv}) $x \leftarrow \text{First}(\text{Max}(\text{Bucket}(N^-(u))))$ if $d^+(x) > \max\{(1 + \lambda) \cdot d^+(u) + \theta, \lfloor \frac{b}{4} \rfloor\}$ Add(\vec{ux}) Delete(\vec{xu}) else for w in next $\lceil \frac{2}{\lambda} \rceil$ neighbors in $N^+(u)$ do $r_w(u) = r(u)$ Move u to bucket $B_{r_w(u)}$ in $N^-(w)$

Overview of techniques. Note that after incrementing (or decrementing) $d^+(u)$, we flip an edge \vec{ux} (or \vec{xu}) whenever the following conditions hold:

$$d^+(u) > \max\left\{(1 + \lambda) \cdot d^+(x) + \theta, \left\lfloor \frac{b}{4} \right\rfloor\right\} \quad (\text{for Insert})$$

$$d^+(x) > \max\left\{(1 + \lambda) \cdot d^+(u) + \theta, \left\lfloor \frac{b}{4} \right\rfloor\right\} \quad (\text{for Delete}).$$

These checks are the same as in Section 4 (and Section 6 for deletions). As a result, the recursive depth of our algorithm is identical to that of Section 4.

The big difference with Section 4, is that during insertions we do not loop over all $x \in N^+(u)$ each time (as that would be too expensive). Instead, we do a round robin scheme where we rely on the fact that if we recently checked the condition for edge \vec{ux} without flipping it, then we need to add many more outgoing edges from u before it violates the actual Invariant θ' . By checking $\lceil \frac{2}{\lambda} \rceil$ edges each time in round-robin order we are guaranteed to revisit \vec{ux} before that happens.

The second difference with Section 4, is that for each vertex u we cannot store a data structure on the in-neighbors of u that uses their actual out-degree. Instead, we bucket the vertices $x \in N^-(u)$ using their out-degree at the time of adding the arc \vec{xu} . The location of x in this data structure is thereby its *perceived* rank $r_u(x)$. Whenever we insert or delete an arc in G^b , we get a recursive call to our insertion and deletion functions that flips a chain of edges. Only the final vertex x_f on this chain changes their actual out-degree. Hence, for this final vertex x_f , we perform round robin over the $\lceil \frac{2}{\lambda} \rceil$ next $w \in N^+(x_f)$ to update the perceived rank of x_f in $N^-(w)$. Again, we can not afford to update all of them.

Recall that we parametrized time according to Definition 1. We show:

LEMMA 5.1. *Let $r_v(u)$ get updated by an Insert or Delete at time s . Let the next update to $r_v(u)$ occur during an Insert or Delete at time t . Then $|d^+(u)_t - d^+(u)_s| \leq \frac{\lambda}{2}d^+(u)_s$ and $|r(u)_t - r(u)_s| \leq 1$.*

Proof. Only out-neighbours to u that exist at time s can be visited by the round-robin procedure before $r_v(u)$ is updated again. Since we visit $\lceil \frac{2}{\lambda} \rceil$ of them per Insert or Delete that changes $d^+(u)$, we can do at most $d^+(u)_s / \lceil \frac{2}{\lambda} \rceil \leq \frac{\lambda}{2}d^+(u)_s$ Inserts or Deletes changing $d^+(u)$ before time t . Thus, since $0 < \lambda < 1$:

$$\begin{aligned} d^+(u)_t &\geq \left(1 - \frac{\lambda}{2}\right)d^+(u)_s > (1 + \lambda)^{-1}d^+(u)_s && \implies r(u)_t \geq r(u)_s - 1 \\ d^+(u)_t &\leq \left(1 + \frac{\lambda}{2}\right)d^+(u)_s < (1 + \lambda)d^+(u)_s && \implies r(u)_t \leq r(u)_s + 1 \end{aligned}$$

□

LEMMA 5.2. *For all edges \vec{uv} at all steps during Insert or Delete, $|r_v(u) - r(u)| \leq 1$.*

Proof. Follows trivially from Lemma 5.1 by the fact that each time it gets updated the true value has changed by at most 1. □

We now apply an argument that we have applied in previous sections, introducing a bit more slack than previously:

LEMMA 5.3. During a Delete(\vec{uv}) at time t , let $x \leftarrow \text{First}(\text{Max}(N^-(u)))$ and

$$d^+(x)_t \leq \max \left\{ (1 + \lambda)(d^+(u)_t - 1) + \theta, \left\lfloor \frac{b}{4} \right\rfloor \right\}.$$

Then for all $w \in N^-(u)$ it must be that:

$$\begin{aligned} d^+(w)_t &\leq (1 + \lambda)^3 \cdot \max \left\{ (1 + \lambda)(d^+(u)_t - 1) + \theta, \left\lfloor \frac{b}{4} \right\rfloor \right\} \\ &\leq \max \left\{ (1 + \eta b^{-1})(d^+(u)_t - 1) + 2\theta, \left\lfloor \frac{b}{2} \right\rfloor \right\}. \end{aligned}$$

Proof. The vertex $x \leftarrow \text{First}(\text{Max}(N^-(u)))$ has the largest perceived rank of all vertices in $N^-(u)$. Thus, the perceived rank $r_u(w)_t$ is at most $r_u(x)_t$. By Lemma 5.2, we now get:

$$r(x)_t \geq r_u(x)_t - 1 \geq r_u(w)_t - 1 \geq r(w)_t - 2 \implies d^+(x)_t \geq (1 + \lambda)^{r(x)_t} \geq (1 + \lambda)^{r(w)_t - 2} \geq (1 + \lambda)^{-3} d^+(w)_t.$$

It follows that $d^+(w)_t \leq (1 + \lambda)^3 \cdot \max \left\{ (1 + \lambda)(d^+(u)_t - 1) + \theta, \left\lfloor \frac{b}{4} \right\rfloor \right\}$. By noting that $(1 + \lambda)^5 \leq (1 + \eta b^{-1}) \leq 2$ we recover the lemma. \square

LEMMA 5.4. If during an Insert at time s , the out-neighbour $x \in N^+(u)_s$ is verified to satisfy $d^+(u)_s + 1 \leq \max \left\{ (1 + \lambda)d^+(x)_s + \theta, \left\lfloor \frac{b}{4} \right\rfloor \right\}$, then at any time t up to and including the next time that we check the constraint we have that:

$$\begin{aligned} d^+(u)_t &\leq (1 + \lambda)^4 \cdot \max \left\{ (1 + \lambda)d^+(x)_t + \theta, \left\lfloor \frac{b}{4} \right\rfloor \right\} \\ &\leq \max \left\{ (1 + \eta b^{-1})d^+(x)_t + 2\theta, \left\lfloor \frac{b}{2} \right\rfloor \right\}. \end{aligned}$$

Proof. If there are no Deletes changing $d^+(x)$ between times s and t , we have $d^+(x)_s \leq d^+(x)_t$ and

$$\begin{aligned} d^+(u)_t &\leq (1 + \lambda) \cdot d^+(u)_s && \text{(By Lemma 5.1)} \\ &\leq (1 + \lambda) \cdot (d^+(u)_s + 1) \\ &\leq (1 + \lambda) \cdot \max \left\{ (1 + \lambda)d^+(x)_s + \theta, \left\lfloor \frac{b}{4} \right\rfloor \right\} && \text{(By our assumption)} \\ &\leq (1 + \lambda) \cdot \max \left\{ (1 + \lambda)d^+(x)_t + \theta, \left\lfloor \frac{b}{4} \right\rfloor \right\} && \text{(Since } d^+(x)_s \leq d^+(x)_t) \\ &\leq \max \left\{ (1 + \eta b^{-1})d^+(x)_t + 2\theta, \left\lfloor \frac{b}{2} \right\rfloor \right\}. \end{aligned}$$

Suppose now that there was a Delete between times s and t that changed $d^+(x)$. Denote by s' the time just after the last such delete finished. It must be that $s < s' \leq t$. Then $d^+(x)_{s'-1} - 1 = d^+(x)_{s'}$. Since after s' , there was no deletion decreasing $d^+(x)$ it must be that $d^+(x)_{s'} \leq d^+(x)_t$ and by Lemma 5.3 for all $w \in N^-(x)_{s'}$:

$$d^+(w)_{s'} \leq (1 + \lambda)^3 \cdot \max \left\{ (1 + \lambda)d^+(x)_{s'} + \theta, \left\lfloor \frac{b}{4} \right\rfloor \right\}$$

In particular, $u \in N^-(x)_{s'}$ and

$$\begin{aligned} d^+(u)_t &\leq (1 + \lambda) \cdot d^+(u)_{s'} && \text{(By Lemma 5.1)} \\ &\leq (1 + \lambda)^4 \cdot \max \left\{ (1 + \lambda)d^+(x)_{s'} + \theta, \left\lfloor \frac{b}{4} \right\rfloor \right\} && \text{(By Lemma 5.3)} \\ &\leq (1 + \lambda)^4 \cdot \max \left\{ (1 + \lambda)d^+(x)_t + \theta, \left\lfloor \frac{b}{4} \right\rfloor \right\} && \text{(Since } d^+(x)_{s'} \leq d^+(x)_t) \\ &\leq \max \left\{ (1 + \eta b^{-1})d^+(x)_t + 2\theta, \left\lfloor \frac{b}{2} \right\rfloor \right\} \end{aligned}$$

\square

LEMMA 5.5. (MAINTAINING INVARIANT θ') *Whenever Algorithms $\overline{7}$ and $\overline{8}$ terminate, they maintain an orientation \vec{G}^b where for each edge \vec{uv} in \vec{G}^b , $d^+(u) \leq \max\{(1 + \eta b^{-1}) \cdot d^+(v) + 2\theta, \lfloor \frac{b}{2} \rfloor\}$.*

Proof. By construction, when calling $\text{Insert}(\vec{uv})$ at time t we always have $d^+(u)_t \leq d^+(v)_t$. As argued in Theorems $\overline{4.2} + \overline{4.1}$, this new edge may never invalidate Invariant θ' between u and v . Now consider the chain of edges that get recursively flipped until we reach the final vertex x_f . The vertex x_f is the only vertex for which $d^+(x_f)_{t+1} = d^+(x_f)_t + 1$. Thus, for all other vertex pairs not including x_f , Invariant θ' is maintained. Since the algorithm terminated at x_f it must be that for all $x \in N^+(x_f)_t$ where the constraint was checked at time t : $d^+(x_f)_{t+1} = d^+(x_f)_t + 1 \leq \max\{(1 + \lambda)d^+(x)_t + \theta, \lfloor \frac{b}{4} \rfloor\} = \max\{(1 + \lambda)d^+(x)_{t+1} + \theta, \lfloor \frac{b}{4} \rfloor\}$. By Lemma $\overline{5.4}$ it follows that Invariant θ' is maintained between x_f and all vertices in $N^+(x_f)_{t+1}$.

The argument for $\text{Delete}(\vec{uv})$ is symmetrical, applying Lemma $\overline{5.3}$ instead. \square

LEMMA 5.6. *Algorithms $\overline{7} + \overline{8}$ spend $\mathcal{O}(\lambda^{-1})$ time before recursing, except for the outermost call which spends $\mathcal{O}(\lambda^{-1} \log n)$ time.*

Proof. Whenever we insert a vertex $v \in N^-(u)$, it is either because we added the edge (u, v) to G (occurring once) or, because we flipped an edge \vec{uv} . In the first case, we may afford spending $\mathcal{O}(\log_{(1+\lambda)} d^+(v)) = \mathcal{O}(\lambda^{-1} \log(bp + \log n)) = \mathcal{O}(\lambda^{-1} \log n)$ time searching through all buckets for the bucket containing v . In the latter case, for Insert we have $\max\{(1 + \lambda)d^+(u)_t, \lfloor \frac{b}{4} \rfloor\} < d^+(v)_t + 1 \leq (1 + \lambda)^4 \max\{(1 + \lambda)d^+(u)_t, \lfloor \frac{b}{4} \rfloor\} + 1$ by Lemma $\overline{5.4}$, and similarly for Delete we have $\max\{(1 + \lambda)(d^+(u)_t - 1), \lfloor \frac{b}{4} \rfloor\} < d^+(v)_t \leq (1 + \lambda)^3 \max\{(1 + \lambda)(d^+(u)_t - 1), \lfloor \frac{b}{4} \rfloor\}$ by Lemma $\overline{5.3}$. Using the pointer from u to the bucket $B_i(u)$ where $i = \lfloor \log_{(1+\lambda)} \max\{(1 + \lambda)d^+(u), \lfloor \frac{b}{4} \rfloor\} \rfloor$, we may insert v into the correct bucket in $\mathcal{O}(1)$ time.

During $\text{Insert}(\vec{uv})$ or $\text{Delete}(\vec{uv})$ we loop over at most $\mathcal{O}(\lambda^{-1})$ elements to change their bucket. By Lemma $\overline{5.1}$, each element changes their position in the data structure by at most 1, which can be done in $\mathcal{O}(1)$ time. \square

Concluding our argument. By Lemma $\overline{5.5}$, our algorithms maintain Invariant θ' at all times. By Lemma $\overline{5.6}$ our algorithms spend $\mathcal{O}(\lambda^{-1})$ time before recursing (except for the outermost call, which uses $\mathcal{O}(\lambda^{-1} \log n)$ time). We now make a case distinction. Either we set $(\theta = 1, b = 1, \eta \in \Theta(\log n))$, or, we set $(\theta = 0, b \in \Theta(\log n), \eta = 3)$. Because our recursive condition in Algorithms $\overline{7} + \overline{8}$ is the same as in Algorithms $\overline{3} + \overline{4}$, we may immediately apply the proofs of Theorem $\overline{4.1} + \overline{4.2}$ to upper bound the recursive depth of our algorithms by $\mathcal{O}(\lambda^{-1} \log \rho)$. Thus, the total time for inserting or deleting a single edge in \vec{G}^b is $\mathcal{O}(\lambda^{-1} \log n + \lambda^{-2} \log \rho) = \mathcal{O}(\lambda^{-2} \log \rho)$. For every update in G , we do $\Theta(b)$ updates in \vec{G}^b . Thus, for both choices of our variables, our algorithms run in time $\mathcal{O}(b \cdot \lambda^{-2} \log \rho)$, and they maintain Invariant θ' for the chosen $\theta \in \{0, 1\}$. Thus, we conclude:

THEOREM 5.1. *Let G be a dynamic graph and ρ be the density of G at update time. We can choose our variables $\theta = 0$, $\eta = 3$, and $b \in \Theta(\log n)$, $b \geq 2$ to maintain an out-orientation \vec{G}^b in worst case $\mathcal{O}(\log^3 n \log \rho)$ time per operation in G , maintaining Invariant $\overline{0}$ for \vec{G}^b with:*

- $\forall v$, the out-degree $d^+(v)$ in \vec{G}^b is at most $\mathcal{O}(b \cdot \rho)$, and
- $\forall u$, the out-degree of u in \vec{G} is at most $\mathcal{O}(\rho)$.

THEOREM 5.2. *Let G be a dynamic graph and ρ be the density of G at time t . We can choose our variables $\theta = 1$, $b = 1$ and $\eta \in \Theta(\log n)$ to maintain an out-orientation $\vec{G}^b = \vec{G}$ in worst case $\mathcal{O}(\log^2 n \log \rho)$ time per update in G such that Invariant $\overline{1}$ holds for \vec{G} . Moreover:*

- $\forall u$, the out-degree $d^+(u)$ in \vec{G} is at most $\mathcal{O}(\rho + \log n)$, (i.e. $\Delta(\vec{G}) \in \mathcal{O}(\rho + \log n)$)

6 Improved amortised algorithms

Previously, we relied upon the fact that vertices in $v \in N^-(u)$ were put in buckets based on their exact out-degree $d^+(v)$. Maintaining these exact values requires $\Omega(\rho)$ update time, and is thus not a suitable option when we aim for polylogarithmic update time. To this end, we store for all edges \vec{uv} a single integer $\phi(u, v)$ which we will call their threshold value. Note that $\phi(u, v) \neq \phi(v, u)$. We base our algorithmic logic and analysis on the threshold value instead. We maintain for all u :

- (a) The value $d^+(u)$ of the current orientation \vec{G}^b ,
- (b) The set $N^+(u)$ as a sorted doubly linked list of linked lists $L_j(u)$. Each $L_j(u)$ contains all $w \in N^+(u)$ with $\phi(u, w) = j$ as a linked list in arbitrary order. The linked lists $L_j(u)$ are stored in a linked list sorted by j . We maintain a pointer to the location $j = d^+(u)$.
- (c) The set $N^-(u)$ in a sorted doubly linked list of buckets $B_j(u)$. Each bucket $B_j(u)$ contains, as a doubly linked list in arbitrary order, all $w \in N^-(u)$ where $j = \lfloor \log_{(1+\lambda)} d^+(u) \rfloor$. The vertex u has a pointer to the bucket $B_i(u)$ with $i = \lfloor \log_{(1+\lambda)} \max\{(1+\lambda)d^+(u), \lfloor \frac{b}{4} \rfloor\} \rfloor$.

Any update in G , invokes Algorithms [1](#)+[2](#). These algorithms now invoke Algorithm [9](#) or [10](#) (instead of [3](#) or [4](#)). These two in turn invoke the normal add and remove functions (Algorithm [5](#)+[6](#)).

Algorithm 9: Insert(\vec{uv}), where $d^+(u) \leq d^+(v)$	Algorithm 10: Delete(\vec{uv})
Add (\vec{uv}) while $\exists x \in N^+(u)$ with $d^+(u) \geq \max\{(1+\lambda) \cdot \phi(u, x), \lfloor \frac{b}{4} \rfloor\}$ do if $d^+(x) + 1 < d^+(u)$ Remove (\vec{ux}) Insert (\vec{xu}) return else $\phi(u, x) = d^+(u)$ Update x in $N^+(u)$ and u in $N^-(x)$	Remove (\vec{uv}) for i in decreasing order do for all $x \in B_i(u)$ do if $d^+(x) > \max\{(1+\lambda)d^+(u) + \theta, \lfloor \frac{b}{4} \rfloor\}$ Add (\vec{ux}) Delete (\vec{xu}) return else if $\phi(x, u) > (1+\lambda)d^+(x)$ $\phi(x, u) = d^+(x)$ Update x in $N^-(u)$ and u in $N^+(x)$ else return

LEMMA 6.1. Suppose that the graph G^b contains \vec{ux} . Then $d^+(u) \leq \max\{(1+\lambda) \cdot \phi(u, x), \lfloor \frac{b}{4} \rfloor\}$.

Proof. Fix some arc \vec{ux} . Whenever the value $\phi(u, x)$ is set, it is set to $d^+(u)$. Thus, we satisfy the inequality. The only risk to the desired inequality is increasing $d^+(u)$, whilst it is bigger than $\lfloor \frac{b}{4} \rfloor$.

Suppose $d^+(u)$ is momentarily increased after adding some arc \vec{uv} . If the inequality for \vec{ux} is violated, then x is eligible for the while loop.

If the while loop processes x , then it will either flip \vec{ux} or resets $\phi(u, x)$. If it flips \vec{ux} , then \vec{ux} is removed from the orientation so there is no inequality to satisfy. Otherwise, we reset $\phi(u, x)$ to $d^+(u)$, which satisfies the inequality.

If the while loop doesn't process x , then it must have selected another vertex $y \in N^+(u)$ and flipped \vec{uy} before processing x . In this case, $d^+(u)$ is restored to its previous value when the inequality for \vec{ux} was satisfied. \square

LEMMA 6.2. Suppose that G^b contains an edge \vec{uz} . Then: $\phi(u, z) \leq \max\{(1+\lambda)^3(d^+(z) + \theta), \lfloor \frac{b}{4} \rfloor\}$.

Proof. Fix an arc \vec{uz} . When the value $\phi(u, z)$ is set, it is set to $d^+(u)$ at a point in time where $d^+(u) \leq \max\{(1+\lambda)d^+(z) + \theta, \lfloor \frac{b}{4} \rfloor\}$. This satisfies the desired inequality. The only risk to the desired inequality is when $d^+(z)$ decreases. Now we perform a case distinction. If $\lfloor \frac{b}{4} \rfloor \geq (1+\lambda)d^+(z) + \theta$, then decreasing $d^+(z)$ did not change the fact that previously $\phi(u, z) \leq \lfloor \frac{b}{4} \rfloor$.

Suppose $d^+(z) \geq \frac{b}{4}$, and that it momentarily decreases after deleting \vec{zy} for some y . We know that $u \in N^-(z)$, so $u \in B_j(z)$ for some integer j . In the loop of **Delete**(\vec{zy}), we consider three cases:

(a): we encounter the vertex $u \in N^+(z)$, without hitting any of the two returns. We know that $\phi(u, z) > (1+\lambda)d^+(u)$ and we set $\phi(u, z)$ to be $d^+(u)$. This decreases $\phi(u, z)$, which means that we continue satisfying the inequality.

(b): we flip an arc \vec{tz} and return, restoring $d^+(z)$ to its original (inequality-satisfying) value.

(c): before reaching case (a), we encounter an arc \vec{tz} for $t \in N^-(z)$ which causes us to hit return (the else in our code). Because we reached this point in the code before case (a) we know that:

1. $\phi(u, z) \leq (1 + \lambda)\phi(t, z)$ (we loop over all buckets $B_i(z)$ in decreasing order and did not encounter the vertex u . Thus, u is either in the same bucket as t or in a lower bucket).
2. $\phi(t, z) < (1 + \lambda)d^+(t)$.
3. $d^+(t) \leq \max\{(1 + \lambda)d^+(z) + \theta, \lfloor \frac{b}{4} \rfloor\} = (1 + \lambda)d^+(z) + \theta$.

Combining these inequalities, we have

$$\phi(u, z) \leq (1 + \lambda)\phi(t, z) \leq (1 + \lambda)^2 d^+(t) \leq (1 + \lambda)^2 \cdot ((1 + \lambda)d^+(z) + \theta)$$

Hence, we recover that $\phi(u, z) \leq (1 + \lambda)^3(d^+(z) + \theta)$. \square

COROLLARY 6.1. *Our amortised algorithms maintain Invariant θ' .*

Proof. We combine Lemma 6.1 and 6.2 to get that for all $z \in N^-(u)$:

$$d^+(u) \leq \max\{(1 + \lambda) \max\{(1 + \lambda)^3(d^+(z) + \theta), \lfloor \frac{b}{4} \rfloor\}, \lfloor \frac{b}{4} \rfloor\} \leq \max\{(1 + \eta b^{-1})d^+(z) + 2\theta, \lfloor \frac{b}{2} \rfloor\}. \quad \square$$

6.1 Running time analysis We now move on to the amortised analysis of the algorithm. At a high level, the idea is as follows. Recall that for each arc \vec{ux} we have a label $\phi(u, x)$ equal to $d^+(u)$ at some point in time. The labels $\phi(u, x)$ guide the data structure by suggesting arcs to flip. When we operate on an arc \vec{ux} based on $\phi(u, x)$, if \vec{ux} is not in fact a good arc to work with, then $d^+(u)$ must have deviated substantially from $\phi(u, x)$, and we reset $\phi(u, x)$ to $d^+(u)$. Loosely speaking, we amortised the effort to relabel \vec{ux} against the change to $d^+(u)$.

LEMMA 6.3. *Adding an arc to G^b takes $\mathcal{O}(\lambda^{-1})$ amortised time.*

Proof. We note that the recursive depth of $\text{Insert}(\vec{uv})$ may be $\mathcal{O}(\rho)$. However, we show that the amortised cost of each edge that we process is not too bad.

Observe that the net effect of adding an arc, after all flips, is to increase the out-degree $d^+(u)$ of a single vertex u by 1.

Now, the running time of adding an arc is proportional to the number of arcs \vec{ux} processed in the while loop over all recursive calls to $\text{Insert}(\vec{uv})$. Each such edge \vec{ux} has $d^+(u) \geq (1 + \lambda)\phi(u, x) + 2\theta$, where $\phi(u, x)$ was set to $d^+(u)$ at a previous point in time. Consequently $d^+(u)$ has increased by at least a $(1 + \lambda)$ -factor since $\phi(u, x)$ was set.

We amortise the time spent processing arcs \vec{ux} for fixed u against the increase to $d^+(u)$. Each time an edge insertion results in increase $d^+(u)$, we pay for $\mathcal{O}(1/\lambda)$ units of work distributed uniformly over $N^+(u)$. That is, each $x \in N^+(u)$ receives $\Omega(1/d^+(u))$ fractional credits. By the time an arc \vec{ux} is processed in the while loop of insertion, \vec{ux} has acquired at least one unit of credit, which pays for the time to process it. \square

LEMMA 6.4. *Removing an arc from G^b takes $\mathcal{O}(\lambda^{-1} \log \rho)$ amortised time.*

Proof. The total running time for a deletion is proportional to the total number of arcs processed in the while loop of $\text{Delete}(\vec{uv})$, over all recursive calls to Delete . Each arc (x, u) processed in the loop (except for the very last one) has one of two outcomes: either it is flipped and we make a recursive call to Delete , or we reset $\phi(x, u)$.

Since our recursive condition is the same as in Algorithm 4, we may immediately apply the proofs for upper bounding the recursive depth for deletions from Theorem 4.2 and 4.1, showing that the recursive depth is $\mathcal{O}(\lambda^{-1} \log \rho)$.

Next we address the number of arcs \vec{xu} where we reset $\phi(x, u)$. We note that $\phi(x, u)$ is only updated when it exceeds $d^+(x)$ by a $(1 + \lambda)$ -factor. In other words, consider the time START when $\phi(x, u)$ was set. At the time END when it is reset to $d^+(x)$, the out-degree of $d^+(x)$ has decreased (by at least a $1 + \lambda$ factor). We consider the approximate rank of $d^+(x)$ at two time steps: a lower bound on the rank when $\phi(x, u)$ was set, and an upper bound for when it is reset during a deletion. Formally, we write: $s = \lfloor \log_{1+\lambda} \phi(x, u) \rfloor$ and $t = \lceil \log_{1+\lambda} d^+(x) \rceil$. Finally we denote $\delta = s - t$. Note that to update our data structure on $N^-(u)$, we need to move $\phi(x, u)$ by $\Theta(\delta)$ buckets. We make a case distinction based on whether $\delta < 3$ or $\delta \geq 3$.

Case 1: $\delta < 3$. In this case when setting $\phi(x, u) = d^+(x)$ we need to move x $\mathcal{O}(1)$ buckets in the data structure on $N^-(u)$. By the time $\phi(u, x)$ is reset, $d^+(x)$ has decreased by at least $(1 + \lambda)^s - (1 + \lambda)^{s-1} \geq \frac{\lambda \phi(u, x)}{(1 + \lambda)}$

since $\phi(u, x)$ was set. The net effect of each deletion (after all flips and recursive calls) is to decrease the degree of a single vertex x by 1. When this occurs, we pay for $4/\lambda$ units of work that are distributed uniformly over $N^+(x)$. Consequently, by the time we reset $\phi(u, x)$ in a call to $\text{Delete}(\vec{uv})$ (for some v), x has already acquired one fractional unit of work to pay for the $\mathcal{O}(1)$ work.

Case 2: $\delta \geq 3$. In this case, the rank of $d^+(x)$ decreased by at least $\Theta(\delta)$ and at least three levels. The net effect of each deletion (after all flips and recursive calls) is to decrease the degree of a single vertex x by 1, at which point we distribute $4/\lambda$ credits over $N^-(x)$. Between time START and END , the out-degree $d^+(x)$ may arbitrarily increase and decrease. However, we can always find a sequence of (not necessarily consecutive) edge deletions $S = \{(\alpha, \beta)_i\}$ such that after deletion $(\alpha, \beta)_i$ in G^b , the out-degree $d^+(x)$ decremented by one, and for any pair of consecutive edge deletions $(\alpha, \beta)_i$ $(\alpha', \beta')_{i+1}$ in S , the out-degree of x at the end of deleting $(\alpha, \beta)_i$ equals the out-degree of x at the start of deleting $(\alpha', \beta')_{i+1}$. Denote by $S^i \subset S$ all deletions in S where after the deletion, the vertex x has a rank $t + i + 1$ for $0 < i < s - t$. For every deletion in S^i , we distribute $4/\lambda$ of units of work over $N^+(x)$.

For all $i \in (0, s - t - 1)$, decreasing the rank of x from $t + i + 1$ to $t + i$ requires exactly $(1 + \lambda)^{t+i+1} - (1 + \lambda)^{t+i} = \lambda(1 + \lambda)^{t+i}$ deletions. Thus, S^i has exactly $(1 + \lambda)^{t+i+1} - (1 + \lambda)^{t+i} = \lambda(1 + \lambda)^{t+i}$. Per definition of S^i , after each deletion, $N^+(x)$ has at most $(1 + \lambda)^{t+i+2}$ out-edges. Thus, whenever we distribute after each deletion $4/\lambda$ credits over all $N^+(x)$, the number of credits per edge C is at least:

$$C \geq \sum_{i=0}^{\delta-2} 4\lambda^{-1} \frac{\# \text{ of deletions in } S^i}{\text{out-degree of } x \text{ during deletions in } S^i} = \sum_{i=0}^{\delta-2} 4\lambda^{-1} \frac{\lambda(1 + \lambda)^{t+i}}{(1 + \lambda)^{t+i+2}} = \sum_{i=0}^{\delta-2} \frac{4\lambda}{\lambda(1 + \lambda)^2} \geq \delta - 2$$

Here, the second-to-last inequality follows from the fact that $\lambda \leq 1$ and thus $\frac{4}{(1+\lambda)^2} \geq 1$. Hence, for $\delta \geq 3$ we have acquired $\mathcal{O}(\delta)$ credits on every edge in $N^+(x)$, which we may use to pay for relocating x by δ -buckets.

□

We may now apply Corollary 3.1 and Corollary 3.2. These set η and b such that $\lambda^{-1} \in \mathcal{O}(\log n)$. We note that for every insertion in G , we insert b edges in G^b . Thus, we conclude:

THEOREM 6.1. *Let G be a dynamic graph and ρ be the density of G at update time. We can choose our variables $\theta = 0$, $\eta = 3$, and $b \in \Theta(\log n)$, $b \geq 2$ to maintain an out-orientation \vec{G}^b in $\mathcal{O}(\log^2 n \log \rho)$ amortized time per operation in the original graph G , maintaining Invariant 0 for \vec{G}^b with:*

- $\forall u$, the out-degree $d^+(u)$ in \vec{G} is at most $\mathcal{O}(\rho + \log n)$, (i.e. $\Delta(\vec{G}) \in \mathcal{O}(\rho + \log n)$)

THEOREM 6.2. *Let G be a dynamic graph and ρ be the density of G at time t . We can choose our variables $\theta = 1$, $b = 1$ and $\eta \in \Theta(\log n)$ to maintain an out-orientation $\vec{G}^b = \vec{G}$ in amortized $\mathcal{O}(\log n \log \rho)$ time per update in G such that Invariant 1 holds for \vec{G} . Moreover:*

- $\forall v$, the out-degree $d^+(v)$ in \vec{G}^b is at most $\mathcal{O}(b \cdot \rho)$, and
- $\forall u$, the out-degree of u in \vec{G} is at most $\mathcal{O}(\rho)$.

7 Obtaining $(1 + \varepsilon)$ Approximations

Finally, we note that we can choose our variables carefully to obtain a $(1 + \varepsilon)$ approximations of the maximum subgraph density or minimum out-degree. Theorem 5.1 implies that, for suitable choices of η and b , we can for any graph G maintain a directed graph \vec{G}^b (where G^b is the graph G with every edge duplicated b times) such that \vec{G}^b maintains Invariant 0. By Theorem 3.1, \vec{G}^b approximates the densest subgraph of G and the minimum out-orientation of G^b (where the approximation factor is dependent on β and η). The running time of the algorithm is $\mathcal{O}(b^3 \cdot \log \alpha)$ where α is the arboricity of the graph. In this section we show that for any $0 < \varepsilon < 1$, we can choose an $\eta > 0$ and a $b \in \mathcal{O}(\varepsilon^{-2} \log n)$ to ensure that \vec{G}^b maintains a:

- $(1 + \varepsilon)$ -approx. of the maximum densest subgraph of G in $\mathcal{O}(\varepsilon^{-6} \log^3 n \log \alpha)$ time.
- $(1 + \varepsilon)$ -approx. of the minimum out-orientation of G^b . This implies an explicit $(2 + \varepsilon)$ -approximation of the minimum out-orientation of G in $\mathcal{O}(\varepsilon^{-6} \log^3 n \log \alpha)$ time.

- $(1 + \varepsilon)$ -approx. of the minimum out-orientation of G^b . Through applying clever rounding introduced by Christiansen and Rotenberg [18] we obtain an explicit $(1 + \varepsilon)$ -**approximation of the minimum out-orientation** of G . By slightly opening their black-box algorithm, we can show that applying their technique does not increase our running time. Thus, our total running time is thus $\mathcal{O}(\varepsilon^{-6} \log^3 n \log \alpha)$.

Obtaining a $(1 + \varepsilon)$ Approximation for Densest Subgraph

COROLLARY 7.1. *Let G be a dynamic graph subject to edge insertions and deletions with adaptive maximum subgraph density ρ . Let G^b be G where every edge is duplicated b times. Let $0 \leq \varepsilon < 1$. We can maintain an orientation \vec{G}^b such that*

$$\rho \leq b^{-1} \cdot \Delta(\vec{G}^b) \leq (1 + \varepsilon)\rho$$

with update time $\mathcal{O}(\varepsilon^{-6} \log^3(n) \log \rho)$ per operation in G .

Proof. We apply Theorem 5.1 in order to maintain an out-orientation satisfying Invariant 0, which by Theorem 3.1 satisfies $\rho(G^b) \leq \Delta(\vec{G}^b) \leq (1 + \gamma)(1 + \eta \cdot b^{-1})^{k_{\max}} \rho(G^b)$. By setting $\gamma = \frac{\varepsilon}{2}$, $\eta = 3$, $b = \lceil \gamma^{-1} \eta \log_{(1+\gamma)} n \rceil \in \mathcal{O}(\varepsilon^{-2} \eta \log n)$, we satisfy the conditions of the Theorem. Since $k_{\max} \leq \log_{1+\gamma} n$, we find that

$$(1 + \eta \cdot b^{-1})^{k_{\max}} \leq e^{\eta b^{-1} \cdot k_{\max}} \leq e^{\gamma} \leq 1 + 2\gamma = 1 + \varepsilon$$

where the last inequality comes from the fact that for $0 \leq x \leq 1$, we have $e^x \leq 1 + 2x$. \square

OBSERVATION 1. *The algorithm of Corollary 7.1 can in $\mathcal{O}(1)$ time per operation, maintain the integers: b^{-1} , $\Delta(\vec{G}^b)$ and thus a $(1 + \varepsilon)$ approximation of the value of the density of G .*

However, to actually output any such realizing subgraph, a bit more of a data structure is needed:

LEMMA 7.1. *For a fully-dynamic graph G , there is an algorithm that explicitly maintains a $(1 + \varepsilon)$ approximation of the maximum subgraph density in $\mathcal{O}(\varepsilon^{-6} \log^3 n \log \alpha)$ total time per operation, and that can output a subgraph realizing this density in $\mathcal{O}(\text{occ})$ time where occ is the size of the output.*

Proof. We use Corollary 7.1 to dynamically maintain an orientation \vec{G}^b in $\mathcal{O}(\varepsilon^{-6} \log^3(n) \log \rho)$ per operation in G . Recall (Theorem 3.1) that we defined for non-negative integers i the sets:

$$T_i := \left\{ v \in V \mid d^+(v) \geq \Delta(\vec{G}^b) \cdot (1 + \eta \cdot b^{-1})^{-i} \right\}$$

(note that since we maintain Invariant 0, the constant c in the previous definition is zero).

Let k be the smallest integer such that $|T_{k+1}| < (1 + \gamma)|T_k|$. Moreover, we showed in Corollary 7.1 that k is upper bounded by $\mathcal{O}(\varepsilon^{-1} \log n)$. We show in Section 3 that (the induced subgraph of the vertex set) T_{k+1} is an approximation of the densest subgraph of \vec{G}^b (and therefore of G). We store the vertices of \vec{G}^b as leaves in a balanced binary tree, sorted on their out-degree. Since every change in G , changes at most $\mathcal{O}(b \log n \log \rho) = \mathcal{O}(\varepsilon^{-2} \log^2 n \log \rho)$ out-degrees in \vec{G} , we can maintain this binary tree in $\mathcal{O}(\varepsilon^{-2} \log^3 n \log \rho)$ additional time per operation in G .

Each internal node of the balanced binary tree stores the size of the subtree rooted at that node. Moreover, we store the maximum out-degree $\Delta(\vec{G}^b)$ as a separate integer, and a doubly linked list amongst the leaves.

After each operation in G , for each integer $i \in [0, \varepsilon^{-1} \log n]$, we determine how many elements there are in T_i as follows: first, we compute the value $V_i = \Delta(\vec{G}^b) \cdot (1 + \eta \cdot b^{-1})^{-i}$. Then, we identify in $\mathcal{O}(\log n)$ time how many vertices have out-degree at least V_i (thus, we determine the size of T_i). It follows that we identify T_k in $\mathcal{O}(\varepsilon^{-1} \log^2 n)$ additional time. We store a pointer to the first leaf that is in T_k . If we subsequently want to output the densest subgraph of G , we traverse the occ elements of T_k in $\mathcal{O}(\text{occ})$ total time by traversing the doubly linked list of our leaves. \square

Related Work While results for densest subgraph [5, 14, 22] can be used to estimate maximum degree of the best possible out-orientation, it is also interesting in its own right. Sawlani and Wang [45] maintain a $(1 - \varepsilon)$ -approximate densest subgraph in worst-case time $\mathcal{O}(\varepsilon^{-6} \log^4 n)$ per update where they maintain an *implicit* representation of the approximately-densest subgraph. They write that they can, in $\mathcal{O}(\log n)$ time, identify the subset $S \subseteq V$ where $G[S]$ is the approximately-densest subgraph and they can report it in $\mathcal{O}(|S|)$

Obtaining an almost $(1 + \varepsilon)$ Approximation for Minimum Out-orientation By Corollary 7.1 we can dynamically maintain for every graph G , a directed graph \vec{G}^b (where each edge in G is duplicated b times) such that the maximum out-degree in \vec{G}^b is at most a factor $(1 + \varepsilon)$ larger than the minimum out-orientation of G^b . For every edge (u, v) in G , we can now store a counter indicating how many edges point (in G^b) from u to v , or the other way around. The naive rounding scheme, states that the edge (u, v) is directed as \vec{uv} whenever there are more edges directed from u to v . For any edge, we can decide its rounding in $\mathcal{O}(1)$ time, thus we conclude:

OBSERVATION 2. *We can maintain for a graph G an orientation \vec{G} where each vertex has an out-degree of at most $(2 + \varepsilon)\alpha$ with update time $\mathcal{O}(\varepsilon^{-6} \log^3(n) \log \rho)$ per operation.*

Obtaining a $(1 + \varepsilon)$ -approximation of the minimum out-orientation of G is somewhat more work. Christiansen and Rotenberg [18] show how to dynamically maintain an explicit out-orientation on G of at most $(1 + \varepsilon)\alpha + 2$ out-edges. In their proofs, Christiansen and Rotenberg [18] rely upon the algorithm by Kopelowitz, Krauthgamer, Porat and Solomon [34]. By replacing the KKPS [34] algorithm by ours in a black-box like manner, we obtain the following:

THEOREM 7.1. *Let G be a dynamic graph subject to edge insertions and deletions. We can maintain an orientation \vec{G} where each vertex has an out-degree of at most $(1 + \varepsilon)\alpha + 2$ with update time $\mathcal{O}(\varepsilon^{-6} \log^3 n \log \alpha)$ per operation in G , where α is the arboricity at the time of the update.*

The proof follows immediately from the proof Theorem 26 by Christiansen and Rotenberg [18] (using Corollary 7.1 as opposed to [34]). For the reader's convenience, we will briefly elaborate on how this result is obtained and how we can apply Corollary 7.1. For the full technical details, we refer to the proof of Theorem 26 in [18].

1. Christiansen and Rotenberg consider a graph G with arboricity α . Moreover, they construct a directed graph \vec{G}^b which is the graph G where every edge in G is duplicated $b \in \mathcal{O}(\varepsilon^{-2} \log n)$ times.² Every operation in G triggers $\mathcal{O}(b)$ operations in \vec{G}^b .
2. On the graph G^b , they run the algorithm by [34] to maintain an orientation of \vec{G}^b where each vertex has an out-degree of at most $\Delta(\vec{G}^b) = (1 + \varepsilon')\alpha \cdot b + \log_{(1+\varepsilon')} n$ for $\varepsilon' = \theta(\varepsilon)$ (for instance $\varepsilon' = \varepsilon/4$ works). The KKPS [34] algorithm uses per operation in G^b .³
 - $\mathcal{O}\left(\left(\Delta(\vec{G}^b)\right)^2\right) = \mathcal{O}\left((1 + \varepsilon)^2 \alpha^2 b^2 + \varepsilon^{-4} \log^2 n\right) = \mathcal{O}(\varepsilon^{-4} \alpha^2 \log^2 n)$ time, and
 - $\mathcal{O}\left(\Delta(\vec{G}^b)\right) = \mathcal{O}\left((1 + \varepsilon)\alpha b + \varepsilon^{-2} \log n\right) = \mathcal{O}(\varepsilon^{-2} \alpha \log n)$ combinatorial changes in \vec{G}^b . (here, a combinatorial change either adds, removes, or flips an edge in \vec{G}^b).
3. Finally, they deploy a clever rounding scheme to transform the orientation \vec{G}^b into an orientation of G where the out-degree of each vertex in \vec{G} is at most a factor $\frac{1}{b}$ the out-orientation of \vec{G}^b , plus two. Thus, they ensure that each vertex has an out-degree of at most:

$$(1 + \varepsilon')\alpha + b^{-1} \log_{1+\varepsilon'} n + 2 \leq (1 + \varepsilon')\alpha + \frac{\varepsilon'^2}{\log n} \cdot \frac{2 \log n}{\varepsilon'} + 2 = (1 + \varepsilon)\alpha + 2$$

since $\alpha \geq 1$ if the graph has at least one edge (otherwise the claim is vacant). They achieve this in $\mathcal{O}(\log n)$ additional time per combinatorial change in \vec{G}^b . Specifically:

- They consider for every edge (u, v) in G its partial orientation (i.e. how many edges in G^b point from u to v or vice versa). If the partial orientation contains sufficiently many edges directed from u to v , the edge in G gets rounded (directed from u to v).

²In [18], Christiansen and Rotenberg choose the duplication constant to be γ and write G^γ .

³Christiansen and Rotenberg deliberately use the adaptive variant of KKPS [34].

- Let H be a (not necessarily maximal) set of edges in G whose direction can be determined in this fashion. They call H a *refinement*. Christiansen and Rotenberg choose H such that in the rounded, directed graph $G - H$ each vertex has an out-degree of at most $(1 + \varepsilon)\alpha$.
- Christiansen and Rotenberg show that H always can be made into a forest. For all edges in H , they no longer explicitly store the b copies in \vec{G}^b . Instead, they store for edges in H their (partial) orientation as an integer in $[0, b]$. The forest H gets stored in a top tree where each interior node stores the minimum and maximum partial orientation of all its children. For any path or cycle in H , they can increment or decrement all orientation integers by 1 in $\mathcal{O}(\log n)$ time by lazily updating these maxima and minima in the top tree. For each edge in H , one can obtain the exact partial orientation in $\mathcal{O}(\log n)$ additional time by adding all lazy updates in the root-to-leaf path of the top tree.
- In addition, they show how to dynamically maintain a 2-orientation on the forest H in $\mathcal{O}(\log n)$ update time per insertion in the forest. Adding the directed edges from the forest to G ensures that each vertex has an out-degree of at most $(1 + \varepsilon)\alpha + 2$.
- For each combinatorial change in \vec{G}^b , they spend $\mathcal{O}(\log n)$ time. Specifically:
 - each combinatorial change in \vec{G}^b may remove an edge from the forest. The edge can be rounded in $\mathcal{O}(1)$ time and removed from the top tree in $\mathcal{O}(\log n)$ time.
 - each combinatorial change may force an edge in G into the refinement and thus possibly creating a cycle.
 - When creating a cycle, the authors augment the cycle such that at least one edge on the cycle may be expelled from the refinement. They (implicitly) increment or decrement all orientation integers along the cycle using the lazy top tree in $\mathcal{O}(\log n)$ total time.
 - Augmenting a cycle causes the out-degree to remain the same for all elements on the cycle. Hence, the Invariants of KKPS [34] (and our Invariant [0]) stay unchanged and the augmentation does not trigger any further operations in \vec{G}^b . Note also that they specifically always leave at least one duplicate edge in each direction, so that no additional data structures need be updated.
- The final edge along the augmented path may subsequently be rounded and added to $G - H$. Thus, spending $\mathcal{O}(\log n)$ time per combinatorial change in \vec{G}^b .

It follows through these three steps that the algorithm in [18] has a running time of:

$$\mathcal{O}\left(b \cdot \left(\left(\Delta(\vec{G}^b)\right)^2 + \Delta(\vec{G}^b) \log n\right)\right) = \mathcal{O}(\varepsilon^{-6} \alpha^2 \log^3 n)$$

Given the results in this paper, we can instead apply our results as follows:

1. We again choose $b \in \mathcal{O}(\varepsilon^{-2} \log n)$. Each operation in G triggers $\mathcal{O}(b)$ operations in \vec{G}^b .
2. We apply Theorem [5.1] (or conversely Corollary [7.1]) to maintain \vec{G}^b such that each vertex has an out-degree of at most $\Delta(\vec{G}^b) = (1 + \theta(\varepsilon))\alpha b$. We proved that this algorithm takes:
 - $\mathcal{O}(b^2 \log \alpha)$ time per operation in \vec{G}^b , but
 - only triggers $\mathcal{O}(b \log \alpha)$ combinatorial changes (edge flips) in \vec{G}^b .
3. Finally, we apply the rounding scheme by Christiansen and Rotenberg which requires $\mathcal{O}(\log n)$ time per *combinatorial change* in \vec{G}^b .

Our total running time is (our algorithm + rounding scheme per combinatorial change):

$$\mathcal{O}(b \cdot b \cdot b \log \alpha + b \cdot b \log \alpha \cdot \log n) = \mathcal{O}(\varepsilon^{-6} \log^3 n \log \alpha).$$

Related Work Historically, four criteria are considered when designing dynamic out-orientation algorithms: the maximum out-degree, the update time (or the recourse), amortised versus worst-case updates, and the adaptability of the algorithm to the current arboricity.

Brodal and Fagerberg [16] were the first to consider the out-orientation problem in a dynamic setting. They showed how to maintain an $\mathcal{O}(\alpha_{\max})$ out-orientation with an amortised update time of $\mathcal{O}(\alpha_{\max} + \log n)$, where α_{\max} is the maximum arboricity throughout the entire update sequence. Thus, their result is adaptive to the current arboricity as long as it only increases. He, Tang, and Zeh [29] and Kowalik [37] provided different analyses of Brodal and Fagerberg's algorithm resulting in faster update times at the cost of worse bounds on the maximum out-degree of the orientations. Henzinger, Neumann, and Wiese [31] gave an algorithm able to adapt to the current arboricity of the graph, achieving an out-degree of $\mathcal{O}(\alpha)$ and an amortised update time independent of α , namely $\mathcal{O}(\log^2 n)$. Kopelowitz, Krauthgamer, Porat, and Solomon [34] showed how to maintain an $\mathcal{O}(\alpha + \log n)$ out-orientation with a worst-case update time of $\mathcal{O}(\alpha^2 + \log^2 n)$ fully adaptive to the arboricity. Christiansen and Rotenberg [18, 17] lowered the maximum out-degree to $(1 + \varepsilon)\alpha + 2$ incurring a worse update time of $\mathcal{O}(\varepsilon^{-6}\alpha^2 \log^3 n)$. Finally, Brodal and Berglin [8] gave an algorithm with a different trade-off; they show how to maintain an $\mathcal{O}(\alpha_{\max} + \log n)$ out-orientation with a worst-case update time of $\mathcal{O}(\log n)$. This update time is faster and independent of α , however the maximum out-degree does not adapt to the current value of α .

8 Applications

In this section, we show how to combine our two trade-offs for out-orientations (theorems 5.1, 5.2 with existing or folklore reductions, obtaining improved algorithms for maximal matching, arboricity decomposition, and matrix-vector product.

8.1 Maximal matchings For our application in maximal matchings, we first revisit the following result. The authors have not seen this theorem stated in this exact generality in the literature, but similar statements appear in [43], [40], and [8]

LEMMA 8.1. (FOLKLORE) *Suppose one can maintain an edge-orientation of a dynamic graph, that has t_u update time, that for each update performs at most r_u edge re-orientations (direction changes), and that maintains a maximum out-degree of $\leq n_o$. Then there is a dynamic maximal matching algorithm⁴ whose update time is $\mathcal{O}(t_u + r_u + n_o)$.*

Proof. [Proof of Lemma 8.1] Each vertex maintains two doubly-linked lists over its in-neighbors (one for the matched, and one for the available in-neighbors) called *in-lists* and a doubly-linked list of its out-neighbors called the *out-list*. When a vertex becomes available because of an edge deletion, it may match with the first available in-vertex if one exists. If no such in-vertex exists, it may propose a matching to its $\leq n_o$ out-neighbors in the out-list, and then match with an arbitrary one of these if any is available. When a vertex v changes status between matched and available, it notifies all vertices in its out-list, who move v between in-lists in $\mathcal{O}(1)$ time. Finally, when an edge changes direction, each endpoint needs to move the other endpoint between in- and out-lists.

The bookkeeping of moving vertices between unordered lists takes constant time. For each edge insertion or deletion, we may spend additionally $\mathcal{O}(n_o)$ time proposing to or notifying to out-neighbors to a vertex, for at most two vertices for each deletion or insertion respectively. \square

With this application in mind, some desirable features of out-orientation algorithms become evident:

- we want the number of out-edges $\Delta(\vec{G})$ to be (asymptotically) low, and
- we want the update time to be efficient, preferably deterministic and worst-case.

Here, a parameter for having the number of out-edges asymptotically as low as possible, can be sparseness measures such as the maximum subgraph density or the arboricity of the graph. An interesting challenge for dynamic graphs is that the density may vary through the course of dynamic updates, and we prefer not to have the update time in our current sparse graph to be affected by a brief occurrence of density in the past. In the work of Henzinger, Neumann, and Wiese, they show how it is possible to adjust to the current graph sparseness in the amortised setting [31]. In this paper, however, we are interested in the case where both the update time is worst-case and the number of re-orientations is bounded. One previous approach to this challenge is to take

⁴When the update time t_u is worst-case, the number of re-orientations r_u is upper bounded by t_u .

a fixed upper bound on the sparseness as parameter to the algorithm, and then use $\log n$ data structures in parallel [45]. Since we want the number of re-orientations to also be bounded, we cannot simply change between two possibly very different out-orientations that result from different bounds on the sparseness. Any scheme for deamortising the switch between structures would be less simple than the approach we see in this paper.

COROLLARY 8.1. *There is a deterministic dynamic maximal matching algorithm with worst-case $\mathcal{O}(\alpha + \log^2 n \log \alpha)$ update time, where α is the current arboricity of the dynamic graph. The algorithm also implies a 2-approximate vertex cover in the same update time.*

Related Work Matchings have been widely studied in dynamic graph models. Under various plausible conjectures, we know that a maximum matching cannot be maintained even in the incremental setting and even for low arboricity graphs (such as planar graphs) substantially faster than $\Omega(n)$ update time [1, 2, 30, 36, 20]. Given this, we typically relax the requirement from maximum matching to maintaining matchings with other interesting properties. One such relaxation is to require that the maintained matching is only *maximal*. The ability to retain a maximal matching is frequently used by other algorithms, notably it immediately implies a 2-approximate vertex cover. In incremental graphs, maintaining a maximal matching is trivially done with the aforementioned greedy algorithm. For decremental⁵ or fully dynamic graphs, there exist a number of trade-offs (depending on whether the algorithm is randomised or deterministic, and whether the update time is worst case or amortised). Baswana, Gupta, and Sen [7] and Solomon [46] gave randomised algorithms maintaining a maximal matching with $\mathcal{O}(\log n)$ and $\mathcal{O}(1)$ amortised update time. These results were subsequently deamortised by Bernstein, Forster, and Henzinger [9] with only a $\text{polylog } n$ increase in the update time. For deterministic algorithms, maintaining a maximal matching is substantially more difficult. Ivkovic and Lloyd [33] gave a deterministic algorithm with $\mathcal{O}((n+m)^{\sqrt{2}/2})$ worst case update time. This was subsequently improved to $\mathcal{O}(\sqrt{m})$ worst case update time by Neiman and Solomon [40], which remains the fastest deterministic algorithm for general graphs.

Nevertheless, there exist a number of results improving this result for low-arboricity graphs. Neiman and Solomon [40] gave a deterministic algorithm that, assuming that the arboricity of the graph is always bounded by α_{\max} , maintains a maximal matching in amortised time $\mathcal{O}(\min_{\beta > 1} \{\alpha_{\max} \cdot \beta + \log_{\beta} n\})$, which can be improved to $\mathcal{O}(\log n / \log \log n)$ if the arboricity is always upper bounded by a constant. Under the same assumptions, He, Tang, and Zeh [29] improved this to $\mathcal{O}(\alpha_{\max} + \sqrt{\alpha_{\max} \log n})$ amortised update time. Without requiring that the arboricity be bounded at all times, the work by Kopelowitz, Krauthgsamer, Porat, and Solomon [34] implies a deterministic algorithm with $\mathcal{O}(\alpha^2 + \log^2 n)$ worst case update time, where α is the arboricity of the graph when receiving an edge-update.

8.2 Dynamic $\Delta + 1$ colouring

LEMMA 8.2. *Suppose one can maintain an edge-orientation of a dynamic graph, that has t_u update time, that for each update performs at most r_u edge re-orientations (direction changes), and that maintains a maximal out-degree of $\leq n_o$. Then there is a dynamic $\Delta + 1$ -colouring algorithm whose update time is $\mathcal{O}(t_u + r_u + n_o)$.*

Proof. For a vertex v , say a colour is *in-free* if no in-neighbor of v has that colour. For a vertex of degree d , keep a doubly linked list of in-free colours from the palette $0, 1, \dots, d$. Keep an array **taken** of size $d + 1$ where the i 'th entry points to a doubly-linked list of in-neighbors of colour i , and an array **free** of size $d + 1$ where the i 'th entry points to the i 'th colour in the list of in-free colours if the i 'th colour is in-free.

The colour of a vertex v is found by finding a colour that is both in-free and out-free: examine the $\leq n_o$ out-neighbors, and use the **free**-array to temporarily move the $\leq n_o$ out-taken colours to a list **out-taken**. Give v an arbitrary free colour from the remaining list, and undo the **out-taken** list. This takes $\mathcal{O}(n_o)$ time, and gives v a colour between 0 and its degree.

When an edge changes direction, this incurs $\mathcal{O}(1)$ changes to linked lists and pointers. When an edge update incurs r_u edge re-orientations, we thus have $\mathcal{O}(r_u)$ such changes. When an edge is inserted/deleted from a properly coloured graph, at most one vertex needs to be recoloured, either because there is a colour conflict, or because its colour number is larger than its degree. This vertex can be recoloured in $\mathcal{O}(n_o)$ time. Thus, the total time per edge insertion or deletion is $\mathcal{O}(t_u + r_u + n_o)$. \square

⁵Maintaining an *approximate maximum matching* decrementally is substantially easier than doing so for fully dynamic graphs. Indeed, recently work by [4] matches the running times for approximate maximum matching in incremental graphs [27]. However, for maximal matching, we are unaware of work on decremental graphs that improves over fully dynamic results.

COROLLARY 8.2. *There is a deterministic dynamic $\Delta + 1$ colouring algorithm with worst-case $\mathcal{O}(\alpha + \log^2 n \log \alpha)$ update time, where α is the current arboricity of the dynamic graph.*

Related Work Previous work presented randomised algorithms with constant amortised update time per edge insertion/deletion [13, 32]. For deterministic algorithms, [12] showed that if one is willing to use $(1 + o(1)) \cdot \Delta$ colours, a $\text{polylog}(\Delta)$ amortised update time is possible. Solomon and Wein [48] extended the algorithm by [12] and further showed that it is possible to maintain an $\alpha \log^2 n$ colouring in $\text{polylog}(n)$ amortised update time.

8.3 Dynamic matrix vector product Suppose we have an $n \times n$ dynamic matrix A , and a dynamic n -vector x , and we want to maintain a data structure that allows us to efficiently query entries of Ax . The problem is related to the Online Boolean Matrix-Vector Multiplication (OMV), which is commonly used to obtain conditional lower bounds [19, 30, 39, 42]. If A is symmetric and sparse, in the sense that the undirected graph G with A as adjacency matrix has low arboricity, then we can use an algorithm for bounded out-degree orientation as a black-box to give an efficient data structure as follows:

LEMMA 8.3. (IMPLICIT IN THM. A.3 IN [34]) *Suppose one can maintain an edge-orientation of a dynamic graph with adjacency matrix A , that has t_u update time, that for each update performs at most r_u edge re-orientations (direction changes), and that maintains a maximal out-degree of $\leq n_o$. Then there is a dynamic matrix-vector product algorithm that supports entry-pair changes to A in $\mathcal{O}(t_u + r_u)$ time, entry changes to the vector x in $\mathcal{O}(n_o)$ time, and queries to the an entry of product Ax in $\mathcal{O}(n_o)$ time.*

Proof. Let each node i store the sum $s_i = \sum_{j \in N^-(i)} A_{ij}x_j$, i.e. the sum of the terms of $(Ax)_i = \sum_{j \in N(i)} A_{ij}x_j$ corresponding to incoming edges at i . Changing entry $A_{ij} = A_{ji}$ in the matrix to or from 0 corresponds to deleting or inserting an edge, which takes t_u time and does at most r_u edge re-orientations. Updating the $\mathcal{O}(1)$ affected sums after inserting, deleting, re-orienting, or re-weighting an edge takes worst case $\mathcal{O}(1)$ time. Any entry update to the matrix A thus takes $\mathcal{O}(t_u + r_u)$ time. When a vector entry x_j changes, we need to update the at most n_o sums $\{s_i\}_{i \in N^+(j)}$, which can be done in worst case $\mathcal{O}(n_o)$ time. Finally, the query for $(Ax)_i$ is computed as $(Ax)_i = s_i + \sum_{j \in N^+(i)} A_{ij}x_j$ in worst case $\mathcal{O}(n_o)$ time. \square

This result is used in [34, Theorem A.3] to give an algorithm for dynamic matrix vector product with running time $\mathcal{O}(\alpha^2 + \log^2 n)$ for updating the matrix, and $\mathcal{O}(\alpha + \log n)$ for updating the vector and for queries.

Combining this theorem with our Theorem 5.2 gives us an algorithm for dynamic matrix vector product with slightly improved time for updating the matrix:

COROLLARY 8.3. *Let A be a symmetric $n \times n$ matrix, and let G be the undirected graph whose adjacency matrix is A . Let x be an n dimensional vector. Then we can support changes to A in $\mathcal{O}(\log^2 n \log \alpha)$ worst case time, changes to x in $\mathcal{O}(\alpha + \log n)$ worst case time, and for each $i \in \{1, \dots, n\}$ we can report $\sum_{j=1}^n A_{ij}x_j$ in worst case $\mathcal{O}(\alpha + \log n)$ time.*

If we instead combine with our Theorem 5.1 we get an algorithm for dynamic matrix vector product with slightly worse time for updating the matrix, but improved time for updating the vector and for queries:

COROLLARY 8.4. *Let A be a symmetric $n \times n$ matrix, and let G be the undirected graph whose adjacency matrix is A . Let x be an n dimensional vector. Then we can support changes to A in $\mathcal{O}(\log^3 n \log \alpha)$ worst case time, changes to x in $\mathcal{O}(\alpha)$ worst case time, and for each $i \in \{1, \dots, n\}$ we can report $\sum_{j=1}^n A_{ij}x_j$ in worst case $\mathcal{O}(\alpha)$ time.*

8.4 Dynamic arboricity decomposition

LEMMA 8.4. ([31, 18]) *Suppose one can maintain an edge-orientation of a dynamic graph, that has t_u update time, and that maintains a maximal out-degree of $\leq n_o$. Then there is an algorithm for maintaining a decomposition into $2n_o$ forests whose update time is $\mathcal{O}(t_u)$.*

Proof. Firstly, as noted in [31, 18]: By assigning the i 'th out-edge of a vertex u to subgraph S_i , one obtains a decomposition into n_o subgraphs, each of which is a pseudoforest. Every vertex has at most one out-edge in each

pseudoforest S_i , and thus, the at most one cycle in each tree of the pseudoforest is a directed cycle according to the orientation.

For maintaining this dynamic pseudoforest decomposition, there is only an $\mathcal{O}(1)$ overhead per edge-reorientation, yielding an $\mathcal{O}(n_o)$ -time algorithm for maintaining n_o pseudoforests.

Then, as noted in [31], we may split each pseudoforest S_i into two forests f_i and f'_i by the following simple algorithm: given a new edge e in f_i , notice that there is at most one edge e' in f_i incident to its head. Now, one can safely insert e in any of the two forests $\{f_i, f'_i\}$ that does not contain this at most one edge e' . Thus, consequently, neither f_i nor f'_i will contain a cycle. \square

Thus, by applying Theorem 5.1, we obtain the following:

COROLLARY 8.5. *There is a deterministic algorithm for maintaining an arboricity decomposition into $\mathcal{O}(\alpha)$ forests, whose worst-case update time is $\mathcal{O}(\log^3 n \log \alpha)$, where α is the current arboricity of the dynamic graph.*

Related Work While an arboricity decomposition of a graph; a partition of its edges into as few forests as possible; is conceptually easy to understand, computing an arboricity decomposition is surprisingly nontrivial. Even computing it exactly has received much attention [25, 26, 21, 44]. The state-of-the-art for computing an exact arboricity decomposition runs in $\tilde{O}(m^{3/2})$ time [25, 26]. In terms of not-exact algorithms there is a 2-approximation algorithm [3, 23] as well as an algorithm for computing an $\alpha + 2$ arboricity decomposition in near-linear time [15].

For dynamic arboricity decomposition, Bannerjee et al. [6] give a dynamic algorithm for maintaining the current arboricity. The algorithm has a near-linear update time. They also provide a lower bound of $\Omega(\log n)$. Henzinger Neumann Wiese [31] provide an $\mathcal{O}(\alpha)$ arboricity decomposition in $\mathcal{O}(\text{poly}(\log n, \alpha))$ time; their result also goes via out-orientation, and they provide a dynamic algorithm for maintaining a $2\alpha'$ arboricity decomposition, given access to any black box dynamic α' out-degree orientation algorithm. Most recently, there are algorithms for maintaining $(\alpha + 2)$ forests in $\mathcal{O}(\text{poly}(\log(n), \alpha))$ update-time [18], and $(\alpha + 1)$ forests in $\tilde{O}(n^{3/4} \text{poly}(\alpha))$ time [17].

References

- [1] Amir Abboud and Søren Dahlgaard. Popular conjectures as a barrier for dynamic planar graph algorithms. In *IEEE 57th Annual Symposium on Foundations of Computer Science, FOCS 2016, 9-11 October 2016, Hyatt Regency, New Brunswick, New Jersey, USA*, pages 477–486, 2016.
- [2] Amir Abboud and Virginia Vassilevska Williams. Popular conjectures imply strong lower bounds for dynamic problems. In *55th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2014, Philadelphia, PA, USA, October 18-21, 2014*, pages 434–443, 2014.
- [3] Srinivasa Rao Arikati, Anil Maheshwari, and Christos D. Zaroliagis. Efficient computation of implicit representations of sparse graphs. *Discret. Appl. Math.*, 78(1-3):1–16, 1997.
- [4] Sepehr Assadi, Aaron Bernstein, and Aditi Dudeja. Decremental matching in general graphs. In Mikolaj Bojanczyk, Emanuela Merelli, and David P. Woodruff, editors, *49th International Colloquium on Automata, Languages, and Programming, ICALP 2022, July 4-8, 2022, Paris, France*, volume 229 of *LIPIcs*, pages 11:1–11:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022.
- [5] Bahman Bahmani, Ravi Kumar, and Sergei Vassilvitskii. Densest subgraph in streaming and mapreduce. *Proc. VLDB Endow.*, 5(5):454–465, 2012.
- [6] Niranka Banerjee, Venkatesh Raman, and Saket Saurabh. Fully dynamic arboricity maintenance. *Theor. Comput. Sci.*, 822:1–14, 2020.
- [7] Surender Baswana, Manoj Gupta, and Sandeep Sen. Fully dynamic maximal matching in $o(\log n)$ update time. *SIAM J. Comput.*, 44(1):88–113, 2015.
- [8] Edvin Berglin and Gerth Stølting Brodal. A Simple Greedy Algorithm for Dynamic Graph Orientation. In *28th International Symposium on Algorithms and Computation (ISAAC 2017)*, volume 92 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 12:1–12:12, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.
- [9] Aaron Bernstein, Sebastian Forster, and Monika Henzinger. A deamortization approach for dynamic spanner and dynamic maximal matching. *ACM Trans. Algorithms*, 17(4):29:1–29:51, 2021.
- [10] Aaron Bernstein and Cliff Stein. Fully dynamic matching in bipartite graphs. In *Automata, Languages, and Programming - 42nd International Colloquium, ICALP 2015, Kyoto, Japan, July 6-10, 2015, Proceedings, Part I*, pages 167–179, 2015.
- [11] Aaron Bernstein and Cliff Stein. Faster fully dynamic matchings with small approximation ratios. In *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016, Arlington, VA, USA, January 10-12, 2016*, pages 692–711, 2016.
- [12] Sayan Bhattacharya, Deeparnab Chakrabarty, Monika Henzinger, and Danupon Nanongkai. Dynamic algorithms for graph coloring. In Artur Czumaj, editor, *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, January 7-10, 2018*, pages 1–20. SIAM, 2018.
- [13] Sayan Bhattacharya, Fabrizio Grandoni, Janardhan Kulkarni, Quanquan C. Liu, and Shay Solomon. Fully dynamic $(\Delta + 1)$ -coloring in $O(1)$ update time. *ACM Trans. Algorithms*, 18(2):10:1–10:25, 2022.
- [14] Sayan Bhattacharya, Monika Henzinger, Danupon Nanongkai, and Charalampos E. Tsourakakis. Space- and time-efficient algorithm for maintaining dense subgraphs on one-pass dynamic streams. In Rocco A. Servedio and Ronitt Rubinfeld, editors, *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing, STOC 2015, Portland, OR, USA, June 14-17, 2015*, pages 173–182. ACM, 2015.
- [15] Markus Blumenstock and Frank Fischer. A constructive arboricity approximation scheme. In *SOFSEM 2020: Theory and Practice of Computer Science - 46th International Conference on Current Trends in Theory and Practice of Informatics, SOFSEM 2020, Limassol, Cyprus, January 20-24, 2020, Proceedings*, volume 12011 of *Lecture Notes in Computer Science*, pages 51–63. Springer, 2020.
- [16] Gerth Stølting Brodal and Rolf Fagerberg. Dynamic representations of sparse graphs. In *In Proc. 6th International Workshop on Algorithms and Data Structures (WADS)*, pages 342–351. Springer-Verlag, 1999.
- [17] Aleksander B. G. Christiansen, Jacob Holm, Eva Rotenberg, and Carsten Thomassen. On Dynamic $\alpha + 1$ Arboricity Decomposition and Out-Orientation. In Stefan Szeider, Robert Ganian, and Alexandra Silva, editors, *47th International Symposium on Mathematical Foundations of Computer Science (MFCS 2022)*, volume 241 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 34:1–34:15, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.
- [18] Aleksander B. G. Christiansen and Eva Rotenberg. Fully-Dynamic $\alpha + 2$ Arboricity Decompositions and Implicit Colouring. In Mikolaj Bojańczyk, Emanuela Merelli, and David P. Woodruff, editors, *49th International Colloquium on Automata, Languages, and Programming (ICALP 2022)*, volume 229 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 42:1–42:20, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.
- [19] Raphaël Clifford, Allan Grønlund, and Kasper Green Larsen. New unconditional hardness results for dynamic and

- online problems. In Venkatesan Guruswami, editor, *IEEE 56th Annual Symposium on Foundations of Computer Science, FOCS 2015, Berkeley, CA, USA, 17-20 October, 2015*, pages 1089–1107. IEEE Computer Society, 2015.
- [20] Søren Dahlgaard. On the hardness of partially dynamic graph problems and connections to diameter. In *43rd International Colloquium on Automata, Languages, and Programming, ICALP 2016, July 11-15, 2016, Rome, Italy*, pages 48:1–48:14, 2016.
- [21] Jack Edmonds. Minimum partition of a matroid into independent subsets. *Journal of Research of the National Bureau of Standards Section B Mathematics and Mathematical Physics*, page 67, 1965.
- [22] Alessandro Epasto, Silvio Lattanzi, and Mauro Sozio. Efficient densest subgraph computation in evolving graphs. In Aldo Gangemi, Stefano Leonardi, and Alessandro Panconesi, editors, *Proceedings of the 24th International Conference on World Wide Web, WWW 2015, Florence, Italy, May 18-22, 2015*, pages 300–310. ACM, 2015.
- [23] David Eppstein. Arboricity and bipartite subgraph listing algorithms. *Inf. Process. Lett.*, 51(4):207–211, August 1994.
- [24] Daniele Frigioni, Alberto Marchetti-Spaccamela, and Umberto Nanni. Fully dynamic shortest paths in digraphs with arbitrary arc weights. *J. Algorithms*, 49(1):86–113, 2003.
- [25] Harold Gabow and Herbert Westermann. Forests, frames, and games: Algorithms for matroid sums and applications. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing, STOC '88*, page 407–421, New York, NY, USA, 1988. Association for Computing Machinery.
- [26] Harold N. Gabow. Algorithms for graphic polymatroids and parametric s-sets. In *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA '95*, page 88–97, USA, 1995. Society for Industrial and Applied Mathematics.
- [27] Fabrizio Grandoni, Stefano Leonardi, Piotr Sankowski, Chris Schwiegelshohn, and Shay Solomon. $(1 + \epsilon)$ -approximate incremental matching in constant deterministic amortized time. In Timothy M. Chan, editor, *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019, San Diego, California, USA, January 6-9, 2019*, pages 1886–1898. SIAM, 2019.
- [28] Fabrizio Grandoni, Chris Schwiegelshohn, Shay Solomon, and Amitai Uzdor. Maintaining an EDCS in general graphs: Simpler, density-sensitive and with worst-case time bounds. In Karl Bringmann and Timothy Chan, editors, *5th Symposium on Simplicity in Algorithms, SOSA@SODA 2022, Virtual Conference, January 10-11, 2022*, pages 12–23. SIAM, 2022.
- [29] Meng He, Ganggui Tang, and Norbert Zeh. Orienting dynamic graphs, with applications to maximal matchings and adjacency queries. In Hee-Kap Ahn and Chan-Su Shin, editors, *Algorithms and Computation - 25th International Symposium, ISAAC 2014, Jeonju, Korea, December 15-17, 2014, Proceedings*, volume 8889 of *Lecture Notes in Computer Science*, pages 128–140. Springer, 2014.
- [30] Monika Henzinger, Sebastian Krinninger, Danupon Nanongkai, and Thatchaphol Saranurak. Unifying and strengthening hardness for dynamic problems via the online matrix-vector multiplication conjecture. In Rocco A. Servedio and Ronitt Rubinfeld, editors, *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing, STOC 2015, Portland, OR, USA, June 14-17, 2015*, pages 21–30. ACM, 2015.
- [31] Monika Henzinger, Stefan Neumann, and Andreas Wiese. Explicit and implicit dynamic coloring of graphs with bounded arboricity. *CoRR*, abs/2002.10142, 2020.
- [32] Monika Henzinger and Pan Peng. Constant-time dynamic $(\Delta + 1)$ -coloring. *ACM Trans. Algorithms*, 18(2):16:1–16:21, 2022.
- [33] Zoran Ivkovic and Errol L. Lloyd. Fully dynamic maintenance of vertex cover. In *Graph-Theoretic Concepts in Computer Science, 19th International Workshop, WG '93, Utrecht, The Netherlands, June 16-18, 1993, Proceedings*, pages 99–111, 1993.
- [34] Tsvi Kopelowitz, Robert Krauthgamer, Ely Porat, and Shay Solomon. Orienting fully dynamic graphs with worst-case time bounds. In Javier Esparza, Pierre Fraigniaud, Thore Husfeldt, and Elias Koutsoupias, editors, *Automata, Languages, and Programming - 41st International Colloquium, ICALP 2014, Copenhagen, Denmark, July 8-11, 2014, Proceedings, Part II*, volume 8573 of *Lecture Notes in Computer Science*, pages 532–543. Springer, 2014.
- [35] Tsvi Kopelowitz, Robert Krauthgamer, Ely Porat, and Shay Solomon. Orienting fully dynamic graphs with worst-case time bounds. In *International Colloquium on Automata, Languages, and Programming*, pages 532–543. Springer, 2014.
- [36] Tsvi Kopelowitz, Seth Pettie, and Ely Porat. Higher lower bounds from the 3sum conjecture. In *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016, Arlington, VA, USA, January 10-12, 2016*, pages 1272–1287, 2016.
- [37] Łukasz Kowalik. Adjacency queries in dynamic sparse graphs. *Inf. Process. Lett.*, 102(5):191–195, May 2007.
- [38] Łukasz Kowalik and Maciej Kurowski. Oracles for bounded-length shortest paths in planar graphs. *ACM Trans. Algorithms*, 2(3):335–363, 2006.
- [39] Kasper Green Larsen and R. Ryan Williams. Faster online matrix-vector multiplication. In Philip N. Klein, editor, *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona*,

- Spain, Hotel Porta Fira, January 16-19*, pages 2182–2189. SIAM, 2017.
- [40] Ofer Neiman and Shay Solomon. Simple deterministic algorithms for fully dynamic maximal matching. *ACM Trans. Algorithms*, 12(1):7:1–7:15, 2016.
 - [41] Krzysztof Onak, Baruch Schieber, Shay Solomon, and Nicole Wein. Fully dynamic MIS in uniformly sparse graphs. *ACM Trans. Algorithms*, 16(2):26:1–26:19, 2020.
 - [42] Mihai Patrascu. Towards polynomial lower bounds for dynamic problems. In Leonard J. Schulman, editor, *Proceedings of the 42nd ACM Symposium on Theory of Computing, STOC 2010, Cambridge, Massachusetts, USA, 5-8 June 2010*, pages 603–610. ACM, 2010.
 - [43] David Peleg and Shay Solomon. Dynamic $(1 + \epsilon)$ -approximate matchings: A density-sensitive approach. In Robert Krauthgamer, editor, *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016, Arlington, VA, USA, January 10-12, 2016*, pages 712–729. SIAM, 2016.
 - [44] Jean-Claude Picard and Maurice Queyranne. A network flow solution to some nonlinear 0-1 programming problems, with applications to graph theory. *Networks*, 12(2):141–159, 1982.
 - [45] Saurabh Sawlani and Junxing Wang. Near-optimal fully dynamic densest subgraph. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing*, pages 181–193, 2020.
 - [46] Shay Solomon. Fully dynamic maximal matching in constant update time. In *IEEE 57th Annual Symposium on Foundations of Computer Science, FOCS 2016, 9-11 October 2016, Hyatt Regency, New Brunswick, New Jersey, USA*, pages 325–334, 2016.
 - [47] Shay Solomon. Dynamic approximate matchings with an optimal recourse bound. *CoRR*, abs/1803.05825, 2018.
 - [48] Shay Solomon and Nicole Wein. Improved dynamic graph coloring. *ACM Trans. Algorithms*, 16(3):41:1–41:24, 2020.