Fully-Dynamic $\alpha + 2$ Arboricity Decompositions and Implicit Colouring

Aleksander B. G. Christiansen
Technical University of Denmark, Lyngby, Denmark

Eva Rotenberg
Technical University of Denmark, Lyngby, Denmark

Abstract

The arboricity α of a graph is the smallest number of forests necessary to cover its edges, and an arboricity decomposition of a graph is a decomposition of its edges into forests. The best near-linear time algorithm for arboricity decomposition guarantees at most $\alpha + 2$ forests if the graph has arboricity α (Blumenstock and Fischer [12]).

In this paper, we study arboricity decomposition for *dynamic graphs*, that is, graphs that are subject to insertions and deletions of edges. We give an algorithm that, provided the arboricity of the dynamic graph never exceeds α, maintains an $\alpha + 2$ arboricity decomposition of the graph in $\text{poly}(\log n, \alpha)$ update time, thus matching the number of forests currently obtainable in near-linear time for static (non-changing) graphs.

Our construction goes via dynamic bounded out-degree orientations, and we present a fully-dynamic algorithm that explicitly orients the edges of the dynamic graph, such that no vertex has an out-degree exceeding $\lfloor(1+\varepsilon)\alpha\rfloor + 2$. Our algorithm is deterministic and has a worst-case update time of $O(\varepsilon^{-6}\alpha^2 \log^3 n)$. The state-of-the-art explicit, deterministic, worst-case algorithm for bounded out-degree orientations maintains a $\beta \cdot \alpha + \log_\beta n$ out-orientation in $O(\beta^2\alpha^2 + \beta\alpha \log_\beta n)$ time [30].

As a consequence, we get an algorithm that maintains an implicit vertex colouring with $4 \cdot 2^\alpha$ colours, in amortised poly-$\log n$ update time, and with $O(\alpha \log n)$ worst-case query time. Thus, at the expense of $\log n$-factors in the update time, we improve on the number of colours from $2^{O(\alpha)}$ to $O(2^\alpha)$ compared to the state-of-the-art for implicit dynamic colouring [27].

2012 ACM Subject Classification Theory of computation → Dynamic graph algorithms

Keywords and phrases Dynamic graphs, bounded arboricity, graph colouring, data structures

Digital Object Identifier 10.4230/LIPIcs.ICALP.2022.42

Category Track A: Algorithms, Complexity and Games

Related Version *Full Version*: `https://arxiv.org/abs/2203.06039` [16]

Funding *Aleksander B. G. Christiansen*: Partially supported by the VILLUM Foundation grant 37507 "Efficient Recomputations for Changeful Problems".
Eva Rotenberg: Partially supported by Independent Research Fund Denmark grants 2020-2023 (9131-00044B) "Dynamic Network Analysis" and 2018-2021 (8021-00249B) "AlgoGraph", and the VILLUM Foundation grant 37507 "Efficient Recomputations for Changeful Problems".

Acknowledgements We thank Krzysztof Nowicki for helpful discussions and ideas.

1 Introduction

Graph colouring is a well-studied problem in computer science and discrete mathematics and has many applications such as planar routing and network optimization [17]. A proper colouring of a graph $G = (V, E)$ on n vertices is an assignment of colours to each vertex in $V(G)$ such that no neighbours receive the same colour. We are interested in minimising the number of colours used. The minimum number of colours that can be used to properly colour G, is called the *chromatic number* of G. It is NP-hard to even approximate the chromatic

number to within a factor of $n^{1-\varepsilon}$ for all $\varepsilon > 0$ [43, 29], but colourings with respect to certain parameters can be efficiently computed. For instance, it is well known that if a graph is uniformly sparse in the sense that we can decompose it into k forests, then we can efficiently compute a colouring: the sparsity of the graph ensures that every subgraph has a vertex of degree at most $2k - 1$, allowing us to compute a $2k$ colouring of the graph in linear time by colouring the vertices in a clever order. The minimum number of forests that the graph can be decomposed into is called the *arboricity* of G. In the past decades, much work has gone into the study of *dynamic algorithms* that are able to efficiently update a solution, as the problem undergoes updates. A general question about dynamic problems is: which (near-) linear-time solvable problems have polylogarithmic updatable solutions?

We study the problem of maintaining a proper colouring of a dynamic graph with bounded arboricity. This class of graphs encompasses, for instance, dynamic planar graphs where $\alpha \leq 3$. Here, the graph undergoes changes in the form of insertions and deletions of edges and one needs to maintain a proper colouring of the vertices with fast update times. We distinguish between two scenarios: one where, as is the case for dynamic planar graphs, we have access to an upper bound on the arboricity α_{max} throughout all updates, and one where we do not. Note that due to insights presented in [39], we can often turn an algorithm for the first scenario into an algorithm for the second by scheduling updates to $O(\log n)$ (partial) copies of the graph, thus incurring only an $O(\log n)$ overhead in the update time.

Barba et al. [6] showed that one cannot hope to maintain a proper, explicit vertex-colouring of a dynamic forest with a constant number of colours in poly-logarithmic update time. Consequently, we cannot maintain explicit colourings where the number of colours depend entirely on α with poly-logarithmic update time - even if we know an upper bound on α. This motivated Henzinger et al. [27] to initiate the study of *implicit* colourings. Here, instead of storing the colours of vertices explicitly in memory, a queryable data structure is provided which after some computations returns the colour of a vertex. If one queries the colours of two neighbouring vertices between updates, the returned colours must differ. Now, we can circumvent the lower bound by using known data structures for maintaining information in dynamic forest to 2-colour dynamic forests in poly-logarithmic update time. Henzinger et al. [27] use this to colour graphs via an *arboricity decomposition* i.e. a decomposition of the graph into forests. They present a dynamic algorithm that maintains an implicit proper $2^{O(\alpha)}$-colouring of a dynamic graph G with arboricity α. Their algorithm adapts to α, but in return it hides a constant (around 40) in the asymptotic notation. Even if one has an upper bound α_{max} on α, the currently best obtainable colouring uses $2^{4(\alpha_{max}+1)}$ colours by combining the arboricity decomposition algorithm from Henzinger et al. [27] with an algorithm of Brodal & Fagerberg [13] that maintains a $2(\alpha_{max} + 1)$ bounded out-degree orientation. Both of these algorithms use a lot of colours. Even for planar graphs with arboricity at most 3, $2^{16} > 60.000$ colours are used. This is quite far from 4 colours, which is always sufficient [3, 38], or the 5 colouring that can be computed in linear time [35, 14, 20].

Dynamic arboricity decompositions. Both colouring algorithms go via dynamic α'-bounded out-orientations. Here, the goal is to orient the edges of the graph while keeping out-degrees low. These are then turned into dynamic $2\alpha'$-arboricity decompositions. By 2-colouring each forest, such a decomposition yields a $2^{2\alpha'}$ colouring. Thus the lower α' is, the fewer colours we use. There has been a lot of work on maintaining dynamic low out-orientations [13, 8, 30, 26, 41], and much of this work aim to improve update complexity by relaxing the allowed out-degree. Motivated by implicit colourings, we provide a different trade-off, providing a lower α' value within polylog(n, α) update time. Specifically, a $\lfloor(1 + \varepsilon)\alpha\rfloor + 2$

dynamic out-orientation with $O(\log^3(n)\alpha^2/\varepsilon^6)$ update-time adaptive to α, and an $\alpha+2$ dynamic arboricity decomposition with $O(\text{poly}(\log n, \alpha_{max}))$ update time, when we have an upper bound α_{max} on the arboricity. Our algorithm maintaining the arboricity decomposition matches the number of forests obtained by the best static algorithm running in near-linear time [12].

These algorithms may also be interesting as they go below the 2α barrier on out-edges and forests respectively. In the static case there exist simple and elegant algorithms computing $2\alpha-1$ out-orientations and arboricity decompositions in linear time [4, 19]. For exact algorithms, the state-of-the-art algorithms spend time $\tilde{O}(m^{10/7})$ [34] or $\tilde{O}(m\sqrt{n})$ [33] for the out-orientation problem, and $\tilde{O}(m^{3/2})$ for the arboricity decomposition problem [21, 22]. Even statically computing an $\alpha+1$ out-orientation [31] resp. an $\alpha+2$ arboricity decomposition [12] takes $\tilde{O}(m)$ time. In the dynamic case, the out-orientation with the lowest bound on the out-degree with $O(\text{poly}(\log n, \alpha))$ update time seem to be the algorithm of Brodal & Fagerberg [13] that achieves $2(\alpha_{max}+1)$ out-degree. In [13] it is also noted that determining exactly the complexity of maintaining a d out-orientation for $d \in [\alpha, 2\alpha]$ is a 'theoretically interesting direction for further research'. We make some progress in this direction by showing how to maintain a $\lfloor(1+\varepsilon)\alpha\rfloor+2$ out-orientation with $\text{poly}(\log n, \alpha, \varepsilon^{-1})$ update time. Thus, if α is a constant, we may carefully choose ε to obtain a polylogarithmic $\alpha+2$ out-orientation.

1.1 Results

Let G be a dynamic graph with n vertices undergoing insertion and deletions of edges, and let α be the current arboricity of the graph; that is α might change, when edges are inserted and deleted. If we at all times have an upper bound α_{max} on α, we say that G is undergoing an α_{max} preserving sequence of updates. We have the following:

▶ **Theorem 1.** *For $1 > \varepsilon > 0$, there exists a fully-dynamic algorithm maintaining an explicit $((1+\varepsilon)\alpha+2)$-bounded out-degree orientation with worst-case insertion time $O(\log^3 n \cdot \alpha^2/\varepsilon^6)$ and worst-case deletion time $O(\log^3 n \cdot \alpha/\varepsilon^4)$*

Using pseudoforest decompositions, we obtain a fully dynamic, implicit colouring algorithm:

▶ **Corollary 2.** *Given a dynamic graph with n vertices, there exists a fully dynamic algorithm that maintains an implicit $2 \cdot 3^{(1+\varepsilon)\alpha}$ colouring with an amortized update time of $O(\log^4 n \cdot \alpha^2/\varepsilon^6)$ and a query time of $O(\alpha \log n)$.*

By moving edges between pseudoforests, we can turn the pseudoforest decomposition into a forest decomposition. This also gives a colouring algorithm using fewer colours.

▶ **Theorem 3.** *Given an initially empty and dynamic graph undergoing an arboricity α_{max} preserving sequence of updates, there exists an algorithm maintaining a $\lfloor(1+\varepsilon)\alpha\rfloor+2$ arboricity decomposition with amortized update time $O(\text{poly}(\log n, \alpha_{max}, \varepsilon^{-1}))$. In particular, setting $\varepsilon < \alpha_{max}^{-1}$ yields $\alpha+2$ forests with an amortized update time of $O(\text{poly}(\log n, \alpha_{max}))$.*

▶ **Corollary 4.** *Given a dynamic graph with n vertices, there exists a fully dynamic algorithm that maintains an implicit $4 \cdot 2^{\alpha}$ colouring with an amortized update of $O(\text{polylog}\, n)$ and a query time of $O(\alpha \log n)$.*

Finally, we modify an algorithm of Brodal & Fagerberg [13] so that it maintains an acyclic out-orientation.

▶ **Theorem 5.** *Given an initially empty and dynamic graph G undergoing an arboricity α_{max} preserving sequence of insertions and deletions, there exists an algorithm maintaining an acyclic $(2\alpha_{max}+1)$ out-degree orientation with an amortized insertion cost of $O(\alpha_{max}^2)$, and an amortized deletion cost of $O(\alpha_{max}^2 \log n)$.*

Table 1 Different dynamic algorithms for maintaining out-orientations. We state the bounds in terms of the arboricity α of the graphs, since many of the results referenced do the same.

Reference	Out-degree	Update time	α
Brodal & Fagerberg [13]	$2(\alpha+1)$	$O(\alpha + \log n)$ am.	fixed
Kopelowitz et al. [30]	$\beta\alpha + \log_\beta n$	$O(\beta^2\alpha^2 + \beta\alpha \log n)$	adaptive
He et al. [26]	$O(\alpha\sqrt{\log n})$	$O(\sqrt{\log n})$ am.	fixed
Berglin & Brodal [8]	$O(\alpha + \log n)$	$O(\log n)$	adaptive
Henzinger et al. [27]	40α	$O(\log^2 n)$ am.	adaptive
Kowalik [32]	$O(\alpha \log n)$	$O(1)$ am.	fixed
New (Thm. 1)	$(1+\varepsilon)\alpha + 2$	$O(\log^3(n)\alpha^2/\varepsilon^6))$	adaptive

1.2 Related Work

Dynamic colouring. Barba et al. [6] give algorithms for the dynamic recolouring problem, and show that c-colouring a dynamic forests incurs $\Omega(n^{\frac{1}{c(c-1)}})$ recolourings per update. Solomon & Wein give improved trade-offs between update time and recolourings and give a deterministic dynamic colouring algorithm parametrized by the arboricity α, using $O(\alpha^2 \log n)$ colours with $O(1)$ amortized update time [41]. Henzinger et al. [27] introduced the study of implicit colouring of sparse graphs in order to circumvent the explicit lower bound of Barba et al. [6]; they maintain an implicit colouring using $2^{O(\alpha)}$ colours, with $O(\log^3 n)$ update time and $O(\alpha \log n)$ query-time. Recently, there has also been a lot of work on the dynamic colouring problem parameterised by the maximum degree [9, 10, 28].

Bounded out-degree orientations. Much of the work with respect to bounded out-degree orientations has gone into either 1) statically computing bounded out-degree orientations with the minimum (or close to it) out-degree [21, 37, 11, 1], or 2) dynamically maintaining bounded out-degree orientations with efficient updates [13, 30, 41, 32, 8], but allowing weaker guarantees on the minimum out-degree (see Table 1). Note that the constant of 40 was extracted from an equation in the proof of Lemma 18 (on page 15) in [27]. The current state-of-the-art for exact, static algorithms have running time $\tilde{O}(m^{10/7})$ [34] and $\tilde{O}(m\sqrt{n})$ [33]. Kowalik [31] also gave an algorithm computing a $\lceil(1+\varepsilon)\alpha\rceil$ out-orientation in $\tilde{O}(m \cdot \varepsilon^{-1})$ time.

Arboricity decompositions. A lot of work has been put into producing efficient static algorithms for computing arboricity decompositions [21, 22, 18, 37] (see [12] for an overview). The fastest static algorithm runs in $\tilde{O}(m^{3/2})$ time [21, 22]. Also approximation algorithms have been studied in the static case. There exists a linear-time 2-approximation algorithm [4, 19]. Furthermore, Blumenstock & Fischer provide an algorithm computing a $\lceil(1+\varepsilon)\alpha\rceil+1$ arboricity decomposition in $\tilde{O}(m \cdot \varepsilon^{-1})$ time. Bannerjee et al. [5] provide an $\tilde{O}(m)$ dynamic algorithm maintaining the exact arboricity α of a dynamic graph, and show a lower bound of $\Omega(\log n)$ for dynamically maintaining arboricity. Henzinger et al. [27] provide a dynamic algorithm for maintaining a $2\alpha'$ arboricity decomposition, given access to any black box dynamic α' out-degree orientation algorithm. (See Table 2.)

Other related work. Motivated by the problem of finding a densest subgraph, Sawlani & Wang [39] gave an (implicit) dynamic approximation algorithm for maintaining a $(1+\varepsilon)\rho$ fractional out-degree orientation, where ρ is the maximum subgraph density. In order to tune the parameters in the algorithm, they use multiple (partial) copies of the same graph, where each copy has a different estimate of the maximum density of the graph.

Table 2 Overview of dynamic algorithms for maintaining arboricity decompositions. Note that applying Lemma 27 to Theorem 5 gives an arboricity decomposition, since the orientation is acyclic.

Reference	Forests	Update time	Uses Lemma from [27]	α
Bannerjee et al. [5]	α	$\tilde{O}(m)$	No	adaptive
Brodal & Fagerberg [13]	$4(\alpha+1)$	$O(\alpha+\log n)$ am.	Yes	fixed
Henzinger et al. [27]	80α	$O(\log^2 n)$ am.	Yes	adaptive
New (Thm. 5)	$2(\alpha+1)$	$O(\alpha^2 \log n)$ am.	Uses Lemma 27	fixed
New (Thm. 3)	$\alpha+2$	$O(\text{polylog}\, n)$ am.	No	adaptive

Computing near optimal out-orientations and arboricity decompositions has also been studied from a distributed point of view. Barenboim & Elkin gave a $(2+\varepsilon)$-approximation in [7]. This has since then been improved to $(1+\varepsilon)$-approximations [24, 25, 42, 23].

1.3 Summary of techniques

It is quite simple to compute 2-approximations of arboricity decompositions and bounded out-degree orientations in the static case: It follows from a Theorem of Nash-Williams [36] that a graph with arboricity α is $2\alpha-1$ *degenerate* i.e. every subgraph of the graph has a vertex of degree at most $2\alpha-1$. By continuously removing a vertex with minimum degree and assigning all remaining edges incident to it as out-edges, one obtains an acyclic $(2\alpha-1)$ out-orientation. By partitioning the edges into $2\alpha-1$ partitions s.t. no vertex has two out-edges in the same partition, one obtains a $(2\alpha-1)$-arboricity decomposition.

However, in order to get static algorithms computing close to optimal out-orientations and arboricity decompositions, one typically formulates the problems as combinatorial optimization problems. These can be difficult to approximate – even in the static case, and thus perhaps even more so dynamically. To achieve the low out-orientation algorithm, we build upon a technique of Sawlani & Wang [39]: if one allows edges to be partially assigned to both endpoints, one gets a relaxed version of the out-orientation problem which we will refer to as the *fractional out-orientation problem*. Here edges are assigned partially to both endpoints and one seeks to minimise the maximum *load* of a vertex, where a vertex's load is the sum of the loads contributed by each edge incident to the vertex.

A novelty in our approach lies in what we call *refinements* of fractional orientations: the edges that assign a substantial load to both endpoints form a subgraph – a refinement – of the original graph. We show how to remove cycles from the refinement without changing the loads of any vertices by reassigning edges along cycles. When the refinement is acyclic, the remaining edges have (almost) decided on which endpoint, they prefer. This ensures that we can naively 'round' all of the edges not in the refinement to become out-edges of the vertices they assign the most load. By 2-orienting the refinement, we obtain an algorithm for maintaining an out-orientation with close to the optimal number of out-edges.

Given an α' out-orientation, it is straighforward to split it into α' pseudoforests i.e. graphs where each component has at most one cycle: partition the edges into α' partitions such that no vertex has out-degree more than one in each partition. Every pseudoforest can be represented as a forest and a matching – simply put exactly one edge from each cycle into the matching. Blumenstock & Fischer [12] show that if one chooses the pseudoforests and the representation of the pseudoforsts in a clever way, then the union of all of the matchings form a forest. Thus by combining these techniques one can go from an α' orientation to an $\alpha'+1$ arboricity decomposition.

It is the same idea that underlies our dynamic arboricity decomposition algorithm: we maintain a refinement and a low out-orientation of the remaining edges as before. Then we partition the oriented edges into pseudoforests and finally we alter these pseudoforests and their representations to arrive at a low arboricity decomposition. The key challenge in making this process dynamic is that the altered pseudoforests must be obtainable by partitioning the current out-orientation; otherwise it is unclear how to maintain the pseudoforests as the graph is updated. In order to achieve this, we modify the approach of Blumenstock & Fischer such that the obtained pseudoforests stay faithful to the underlying orientation. However to do so, we have to alter the underlying orientation to accommodate our choice of pseudoforests. To be able to alter the orientation maintained by the out-orientation algorithm, we have to be careful to keep auxiliary datastructures updated and ensure that certain invariants are maintained. To achieve this, we show how to update the datastructures lazily, and we use a potential based argument to show that we can afford to maintain the required invariants.

Paper Outline. In Section 2, we first show how to 2-orient forests and then we recall the techniques of Sawlani & Wang [39] and Kopelowitz et al. [30]. In order to maintain the refinement, we need to represent some parts of the fractional out-orientation implicitly, hence in Section 2 we also make precise exactly how the fractional out-orientation can be accessed. In Section 3 we introduce refinements and show how to maintain them dynamically thus obtaining Theorem 1 and Corollary 2. In Section 4 we show Theorem 3 and Corollary 4. We begin by briefly discussing our approach, before we in Section 4.1 recall the techniques of Henzinger et al. [27] and Blumenstock & Fischer [12]. In Sections 4.2 to 4.7 we describe our new dynamic algorithm for maintaining an arboricity decomposition with close to the optimal number of forests. Finally, Section 5 is dedicated to Theorem 5. Due to space-constraints some proofs are left out (they can be found in the full-version [16]).

2 Preliminaries & Warm-up

Nash-Williams [36] showed that the arboricity α of a graph G satisfies $\alpha = \lceil \max_{J \subset G} \frac{|E(J)|}{|V(J)|-1} \rceil$. The *pseudoarboricity* α_p is the minimum number of pseudoforests that the edges of G can be partitioned into. The *maximum (subgraph) density* or the *fractional pseudoarboricity* ρ is defined as $\rho = \max_{J \subset G} \frac{|E(J)|}{|V(J)|}$. We have $\alpha_p = \lceil \rho \rceil$ [37]. Note that α and α_p are numerically very close, and that $\alpha_p(G)$ is the lowest maximum out-degree achievable when orienting the edges of an undirected graph G [31]. For an undirected graph G and a vertex $v \in V(G)$, we let $d(v)$ be the degree of v and $N(v)$ the neighbourhood of v. If G is directed, $d^+(v)$ denotes the out-degree of v and $N^+(v)$ the out-neighbourhood of v.

Explicit 2-out orientation of dynamic forests. We begin by considering the simpler problem of orienting the edges of dynamic forests so as to minimise the maximum out-degree of vertices. If we want an implicit out-orientation of a dynamic forest H, we can root each tree in H arbitrarily, and get an out-degree of 1 upon query-time using data structures for maintaining information in a dynamic forest (see for example [2, 40]). However, if we want the out-orientation to be explicit, there is a naive lower bound for maintaining a 1-out orientation: Take a path of length n. Deleting the edge between vertex $n/2$ and $n/2 + 1$, yields two sub-paths of length roughly $n/2$. No matter the 1-orientation of these paths, we can reconnect them so as to necessitate $\Omega(n)$ reorientations to restore a 1-orientation. This may be repeated to defy even the hope of an improved amortised analysis.

On the contrary, to obtain a 2-orientation, one can use dynamic heavy path decompositions to obtain an $O(\log(n))$ update time algorithm via orienting light edges towards the root, and heavy edges arbitrarily. For a rooted-tree T, we have the notion of parents and children of the vertices. The *parent* of v is the first vertex from v on the v-to-root path in T. The *children* of v are all neighbours of v that are not the parent of v. A *heavy child* w of v is then a child of v such that the sub-tree of T rooted at w contains more than half of the vertices of the sub-tree rooted at v. The heavy children in T induces *heavy edges* going from a vertex to its heavy child, and *light edges* going from a vertex to its non-heavy children. Every root-to-leaf path then contains at most $O(\log n)$ light edges. The heavy edges form the desired paths, and the light edges can be assigned to the endpoint that is a child of the other endpoint. Sleator and Tarjan [40] showed how to maintain such a heavy-light decomposition in $O(\log n)$ worst case update time, and with $O(1)$ overhead one can keep all light edges oriented towards the root. As such, we have:

▶ **Lemma 6.** *There exists a fully-dynamic algorithm maintaining an explicit 2-out orientation of an n-vertex dynamic forest with $O(\log n)$ worst-case update time.*

Fractional Out-degree Orientations. We will obtain a low-bounded out-degree orientation by deterministically rounding a *fractional out-degree orientation*. Here, the orientation problem is relaxed so that the edges are allowed to be assigned partially to each end-point, and the goal is to compute an orientation such that the maximum total load assigned to a vertex is minimized. A formal definition is as follows:

▶ **Definition 7.** *A* fractional α'-bounded out-degree orientation O *of a graph G is a pair of variables $X_e^u, X_e^v \in [0,1]$ for each edge $e = uv \in E(G)$ s.t. the following holds:*

1. $\forall e = uv \in E(G)$: $X_e^u + X_e^v = 1$
2. $\forall v \in V(G)$: $s(v) = \sum_{e:v\in e} X_e^v \leq \alpha'$

If furthermore $X_e^u, X_e^v \in \gamma^{-1} \cdot \mathbb{Z}$ for all $e \in E(G)$, we say that O is a (γ, α')-orientation.

In particular, an α'-bounded out-degree orientation is just a $(1, \alpha')$-orientation. We think of $s(v)$ as the load on vertex v, and α' as an upper bound on the allowed vertex load. The γ-parameter underlines the fact that we wish to discretise the fractional loads on edges to rational loads. If one does so in a symmetric manner for each edge, one can view a (γ, α')-orientation of a graph G as a $(1, \gamma\alpha')$-orientation of G^γ, where we define G^γ to be G, where every edge is replaced by γ copies. For an edge $e = uv \in E(G)$, we denote by B_e the bundle of γ edges representing e in G^γ. If G^γ is oriented, we denote by B_e^u the bundle of edges oriented $u \to v$. Since the copies of e in B_e are identical, we only care about the size of B_e^u, and not which copies of e it contains. Hence:

▶ **Observation 8.** *For a graph G, there is a natural bijection (up to symmetry) between $(1, \gamma \cdot \alpha')$ orientations of G^γ and (γ, α')-orientations of G.*

In light of this observation, we shall use these two descriptions interchangeably, and in some cases we shall refer to the same orientation as being both a $(1, \gamma\alpha')$-orientation of G^γ and a (γ, α')-orientation of G. We follow the approaches of Sawlani & Wang [39] and Kopelowitz et al. [30], so we repeat the following:

▶ **Definition 9** ([39])**.** *Given a $(1, \alpha')$-orientation of a graph G^γ, we say that an edge $u \to v \in E(G)$ is* η-valid *if $s(u) - s(v) \leq \eta$ and* η-invalid *otherwise. If also $s(v) - s(u) \leq \eta$, we say that $e = uv \in G$ is* doubly η-valid*. Furthermore, if $s(v) - s(u) \leq -\eta/2$ we say that e is an η-tight out-edge of u and an η-tight in-edge of v.*

Note that if $u \to v$ is η-invalid, then $s(u) - s(v) > \eta$ and so $-\eta > s(v) - s(u)$, so uv is η-tight. Note that typically $\eta = 1$.

▶ **Definition 10** ([39] Def. 3.5). *A* maximal η-tight chain from v *is a path of η-tight edges $v_0v_1, \dots v_{k-1}v_k$, such that $v_0 = v$ and v_k has no η-tight out-edges.*

A maximal η-tight chain to v *is a path of η-tight edges $v_0v_1, \dots v_{k-1}v_k$, such that $v_k = v$ and v_0 has no η-tight in-edges.*

▶ **Lemma 11** (Implicit in [39]). *Inserting an η-valid edge oriented $u \to v$ and reorienting a maximal η-tight chain from u will η-invalidate no η-valid edges.*

Deleting an edge oriented $u \to v$ and reorienting a maximal η-tight chain to u will η-invalidate no η-valid edges.

▶ Remark 12. Note that a maximal η-tight chain has length at most $\frac{2 \cdot \max_v s(v)}{\eta}$. Indeed, each time we follow an η-tight out-edge the load on the vertex increases by at least $\eta/2$.

If every edge is η-valid, Sawlani & Wang say that the orientation is *locally η-stable*. Kopelowitz et al. show the following guarantees for locally 1-stable orientations, where we, for ease of notation, define $\Delta^+ := (1+\varepsilon)\alpha\gamma + \log_{(1+\varepsilon)}(n)$:

▶ **Lemma 13** (Implicit in [30]). *If every edge in G^γ is 1-valid, then $\max_v s(v) \leq \Delta^+$.*

Implicit orientations. We are interested in maintaining a fractional out-degree orientation in which the fractional orientation of edges allow us to 'round' the fractional orientation to a low out-degree orientation. We are interested in two properties: first of all the maximum load of a vertex should be low, and second of all many of the edges should have either X_e^u or X_e^v close to 1, so that a naive rounding strategy does not increase the load of a vertex by much. By Lemma 13, if we ensure that the orientation is locally 1-stable, then we get an upper bound on the maximum vertex load. In order to ensure the second property, we redistribute load along cycles without breaking local stability. Our algorithm has two phases. A phase for inserting/deleting edges in a manner that η-invalidates no edges, thus ensuring the first property, and a second phase for redistributing load along edges in order to ensure that the orientation also has the second property. In order for these two phases to work (somewhat) independently, we think of each phase as having implicit access to the orientation; that is the insertion/deletion algorithm might have to pay a query cost in order to identify the precise fractional load of an edge or neighbourhood of a vertex.

▶ **Definition 14.** *An algorithm on an n vertex dynamic and oriented graph has implicit $(|L|, q(n))$ access to an orientation, if it has access to:*

1. *Operations for querying and changing fractional loads of edges in $O(\log n)$ time.*
2. *A query that returns a list containing a superset of all neighbours of a vertex that have changed status as in- or out-neighbour, since the last time the query was called on this vertex. The list should have length $\leq |L|$ and the query should run in $O(q(n))$ time.*

Implicitly Accessing Orientations. In this section, we outline how to modify the algorithm of Kopelowitz et al. [30] to run on G^γ and to support implicit access to the orientation. The ideas presented here are not new; they arise in [30] and [39], but we present them for completeness.

We think of the algorithm as being run on G^γ for some γ to be specified later. We think of each edge $e \in G$ as γ copies in G^γ, but in practice we only store e along with counters $|B_e^u|$, $|B_e^v|$ denoting the number of copies oriented in each direction. Now, we wish to run the

algorithm from [30] in order to insert/delete each copy of an edge one-by-one. This algorithm inserts/deletes a copy of an edge in G^γ using Lemma 11 with $\eta = 1$. We identify a tight chain from u by continuously looking at all out-neighbours and following tight out-edges, until the chain becomes maximal. We use max-heaps, stored at each vertex, to identify maximally tight chains to u. Since we only have implicit access to the orientation, we have to first process the list of possible changes to in- and out-neighbours before trying to identify the next tight edge. Furthermore, when we reorient said chains, we have to access the fractional load of each edge on the chain, before changing it. Hence, we have:

▶ **Theorem 15** (implicit in [30]). *Given implicit $(|L|, q(n))$ access to an orientation with $\max_v s(v) \leq \Delta^+$, there exists an algorithm that can insert and delete edges from the orientation without creating any new 1-invalid edges. The algorithm has worst-case insertion time of $O(\gamma \cdot \Delta^+(\Delta^+ + \log(n)(|L|+1) + q(n)))$ and a worst case deletion time of $O(\gamma \cdot \Delta^+(\log(n)(|L|+1) + q(n)))$.*

▶ Remark 16. Each insertion/deletion of a copy of an edge in G^γ with $\max_v s(v) \leq \Delta^+$ changes the load of at most $O(\Delta^+)$ edges. Indeed, we only change the load of edges on tight chains (and potentially one new edge), so the statement follows from Remark 12.

Scheduling Updates. Some of the algorithms rely on upper bounds on the arboricity. This is, however, not as limiting a factor as one might think, if we are willing to settle for implicit algorithms. In this section we describe how to use the algorithm of Sawlani & Wang [39] to schedule updates to $O(\log n)$ different copies of a graph such that each copy satisfies different density constraints. Here, we describe the main ideas behind the algorithm, and in the appendix of the full version [16], we paraphrase the ideas in more details.

Sawlani & Wang [39] maintain a fractional out-orientation of a graph G by using an algorithm similar to Theorem 15 to insert and delete edges in G^γ. By allowing η to scale with the maximum density ρ of G, they are able to make the update time independent of the actual value of ρ, provided that they have accurate estimates of ρ. By using $O(\log n)$ copies of G – each with different estimates ρ_{est} of ρ, they are able to at all times keep the copy where $\rho_{est} \leq \max_v s(v) < 2\rho_{est}$ fully updated. They call this copy the *active* copy. Similar to Remark 12 they observe that one can safely insert an edge uv in a copy where at least one of $s(u)$ or $s(v)$ is below $2\rho_{est}$. If, however, this is not the case, one cannot afford to update the copy. Sawlani & Wang resolve this issue by scheduling the updates so that they are only performed, when we can afford to do them. We can use this algorithm as a scheduler for our algorithms: We also run our algorithm on $O(\log n)$ copies of G. Whenever the algorithm from Theorem 1.1 in [39] has fully inserted or deleted an edge in a copy, we insert or delete the edge in our corresponding copy. Whenever our algorithm is queried, we then use the structure from the currently active copy to answer the query. Hence, we have:

▶ **Theorem 17** (Implicit in [39] as Theorem 1.1). *There exists a fully dynamic algorithm for scheduling updates that at all times maintains a pointer to a fully-updated copy with estimate ρ_{est} where $(1-\varepsilon)\rho_{est}/2 \leq \alpha(G) < 4\rho_{est}$. Furthermore, the updates are scheduled such that a copy G' with estimate ρ' satisfies $\alpha(G') \leq 4\rho'$. The algorithm has amortised $O(\log^4(n)/\varepsilon^6)$ update times.*

3 Dynamic Low Out-Orientations

In this section, we work towards the second goal: an orientation where many edges assign most of their load to one endpoint. To realise this, we introduce refinements of fractional orientations and show how to dynamically maintain them. The basic idea is to maintain

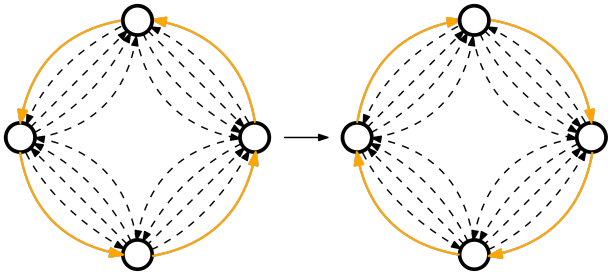

Figure 1 Inverting a cycle. We can reorient copies of edges forming a directed cycle without changing the load of any vertex. By reorienting edges in the refinement, at least one edge on the cycle becomes almost completely oriented towards one endpoint, and hence it can be expelled.

an orientation such that the fractional load of each vertex is small and such that the edges, that distribute their loads somewhat equally between both endpoints, form a forest. This property is nice, if we want to transform our orientation to a bounded out-degree orientation, since all edges outside of the forest almost already have decided on an orientation, and we can 2-orient trees using techniques from Section 2. Definition 18 formalises this idea:

▶ **Definition 18.** *Let O be a (γ, α')-orientation of a graph G. Then H is a (δ, μ)-refinement of G wrt. O if:*

1. $V(H) = V(G)$
2. *For all $e = uv \in E(G) : X_e^u, X_e^v \in (\delta, 1-\delta)$ implies that $e \in H$.*
3. *If $e \in H$, then $X_e^u, X_e^v \in [\delta - \mu, 1 - \delta + \mu]$*

The basic idea behind our algorithm is to maintain a refinement that is a forest. Whenever a cycle C occurs in the refinement, we can redistribute the fractional loads along the cycle so as to not change $s(v)$ for any $v \in C$, but such that an edge of C does not satisfy condition 2. in Definition 18 (see Figure 1). Thus we can remove this edge and again obtain an acyclic refinement with respect to this new orientation. Hence, we have the following observation:

▶ **Observation 19.** *Suppose $1 > \delta > \gamma^{-1} + \mu \geq 2\gamma^{-1} > 0$. Let H be (δ, μ)-refinement of a graph G wrt. some (γ, α')-orientation O of G. Then there exists a (δ, μ)-refinement of G, say H', wrt. some (γ, α')-orientation O' of G, such that H' is a forest.*

Note that if every edge of a graph G is η-valid, then an edge $e = uv \in G$ can only distribute its load somewhat evenly between u and v, if $s(u)$ and $s(v)$ are approximately the same. This implies that e is actually doubly η-valid:

▶ **Observation 20.** *If every edge in G^γ is η-valid, then every edge of a (δ, μ)-refinement H of G with $1 > \delta > \gamma^{-1} + \mu \geq 2\gamma^{-1}$ is doubly η-valid.*

Since the redistribution of fractional load of edges along a cycle does not change the load $s(v)$ of any vertex v, performing the redistribution from Observation 19 η-invalidates no edges.

3.1 The algorithm

As outlined earlier, our algorithm has two phases. In the first phase, we will insert and delete edges without η-invalidating any edges. We do this using the algorithm from Section 2. In the second phase, we examine all of the edges, whose fractional load was altered in phase 1. These edges might need to enter or exit H, depending on their new load. If such an insertion in H creates a unique cycle, we remove it as described in Observation 19. More precisely, the algorithm works as follows:

1. Insert (delete) γ copies of e into G^γ one at a time using a phase I algorithm from Section 2. Whenever a copy of an edge $f \in E(G)$ is reoriented in phase I, we push f onto a stack Q. If e is deleted in G, we also remove e from H.
2. When all γ copies of e are inserted, we set $g = uv = \texttt{pop}(Q)$ and update H as follows until Q is empty:
   - If $g \in H$ and $X_g^u \in (\delta, 1-\delta)$, we update the weight of g in H to match that of G^γ.
   - If $g \in H$ and $X_g^u \notin (\delta, 1-\delta)$, we remove g from H.
   - If $g \notin H$ and $X_g^u \in (\delta, 1-\delta)$, we push g onto a new stack S.
   - If $g \notin H$ and $X_g^u \notin (\delta, 1-\delta)$, we do nothing.
3. After processing all of Q, H together with the edges in S form a (δ, μ)-refinement of G. We now process each edge $h = uv \in S$ as follows:
   - If u, v are not in the same tree in H, we insert h into H.
   - Otherwise, u, v are in a unique cycle C in H. We update the weights along C, locate an edge wz along C with $X_{wz}^w, X_{wz}^z \notin (\delta, 1-\delta)$ and remove it from H. If $uv \neq wz$, we insert uv into H.
   - Finally, we update B_{wz}^w, B_{wz}^z in $G - H$ to match the weights wz had in H.

Since only edges from Q can enter S, we have the following Observation:

▶ **Observation 21.** *Let $S_{\max}$ and $Q_{\max}$ denote the maximum size of the stacks above during an insertion or a deletion. Then we have $S_{\max} \leq Q_{\max} \leq T$, where T is the total number of edges whose fractional orientation are altered during an insertion or a deletion.*

Furthermore, Observations 19 and 20 and Theorem 15 imply the invariants:

▶ **Invariant 22.** *Under the orientation induced by H for edges in $E(H)$ and by $G - H$ for edges in $E(G - H)$, every edge in $E(G)$ is η-valid.*

▶ **Invariant 23.** *H is both a (δ, μ)-refinement and a forest.*

3.2 Implementing updates

Since we maintain the invariant that H is a forest, we can use data structures for maintaining information in fully dynamic forests to store and update H:

▶ **Lemma 24** (Implicit in [2]). *Let F be a dynamic forest in which every edge $e = wz$ is assigned a pair of variables $X_e^w, X_e^z \in [0, 1]$ s.t. $X_e^w + X_e^z = 1$. Then there exists a data structure supporting the following operations, all in $O(\log |F|)$-time:*

- *link$(u, v, X_{uv}^u, X_{uv}^v)$: Add the edge uv to F and set $X_{uv}^u, X_{uv}^v = 1 - X_{uv}^u$ as indicated.*
- *cut(u, v): Remove the edge uv from F.*
- *connected(u, v): Return* true *if u, v are in the same tree, and* false *otherwise.*
- *add_weight(u, v, x): For all edges wz on the path $u \ldots wz \ldots v$ between u and v in F, set $X_{wz}^w = X_{wz}^w + x$ and $X_{wz}^z = X_{wz}^z - x$.*
- *min_weight(u, v): Return the minimum X_{wz}^w s.t. wz is on the path $u \ldots wz \ldots v$ in F.*
- *max_weight(u, v)): Return the maximum X_{wz}^w s.t. wz is on the path $u \ldots wz \ldots v$ in F.*

Note that using non-local search as described in [2], one can also locate the edges of minimum/maximum weight in $O(\log |F|)$-time. The lemma also shows that we can process an edge in Q in $O(\log n)$-time.

▶ **Observation 25.** *We can access and change the fractional load of $e \in H$ in time $O(\log n)$. We can do the same for $e \in G - H$ in $O(1)$ time, since these loads are not stored in top trees.*

To process an edge $uv \in S$ creating a cycle C in the (δ, μ) refinement H, we do as follows. Depending on the argument minimizing $l(C) = \mu + \min\{min\_weight(u,v) - \delta, 1 - \delta - max\_weight(u,v), X^u_{uv} - \delta, 1 - \delta - X^v_{uv}\}$, we either add or subtract $l(C)$ to every edge in C. We determine and remove the edge that minimized $l(C)$ from H. Thus we can process an edge in S in $O(\log n)$-time. Finally, if $\mu > \gamma^{-1}$ then every edge in $S \cup H$ has at least one copy in G^γ pointing in each direction both before and after the inversion of a cycle. Hence, no vertex receives any new in- nor out-neighbours. Since inverting a cycle does not change the load of any vertex, we need not update any priority queues for the insertion/deletion algorithm. Hence, we do not have to return any lists and so $|L| = q(n) = 0$.

3.3 Conclusions

▶ **Theorem 26.** *Suppose $1 > \delta > \gamma^{-1} + \mu > 2\gamma^{-1} > 0$, $\varepsilon > 0$. Then, there exists a dynamic algorithm that maintains a $(\gamma, (1+\varepsilon)\alpha + \gamma^{-1}\log_{(1+\varepsilon)} n)$-orientation of a dynamic graph G with arboricity α as well as a (δ, μ)-refinement H of G wrt. this orientation such that H is a forest. The fractional orientation of an edge can be computed in time $O(\log n)$, insertion takes worst-case $O(\gamma \cdot (\Delta^+)^2)$ time and deletion takes worst-case $O(\gamma \cdot \Delta^+ \cdot \log(n))$ time.*

Proof. Apply Theorem 15 for insertion/deletion. Note that $|L| = 0$ and $q(n) = 0$. The time spent repairing H after each insertion/deletion is in $O(\gamma\Delta^+ \log n)$ by Remark 16 and Observations 21 and 25, since we can process an edge from both Q and S in $O(\log n)$ time. Finally, Observation 20 and the Invariants 22 and 23 show correctness of the algorithm. ◀

Now tuning the parameters of Theorem 26, rounding edges in $G - H$ and 2-orienting H yields Theorem 1. By naively rounding $G - H$ in Theorem 26 and splitting the orientation using Lemma 27, we get an algorithm for dynamically maintaining a decomposition into $\lfloor(1+\varepsilon)\alpha'\rfloor$ pseudoforests and a single forest. Applying the colouring techniques from the full version yields Corollary 2.

4 Forests

We begin this section by outlining the main ideas for turning a dynamic low out-orientation into a dynamic low arboricity decomposition. Given a dynamic α'-bounded out-degree orientation, one can, with very little overhead, split it into α' 1-bounded out-degree orientations using a (slight modification) of an algorithm by Henzinger et al. [27]. Now, given this dynamic pseudoforest partition, we wish to apply the ideas of Blumenstock & Fischer [12] in order to turn the α' pseudoforests into $\alpha' + 1$ forests. The main technical challenge of making this process dynamic is the following: the algorithm from [27] relies heavily on each vertex having out-degree no more than 1 in each pseudoforest. However, the approach of Blumenstock & Fischer [12] is to move edges between pseudoforests, showing no regards as to why an edge was placed in a pseudoforest to begin with. Hence, if one naively applies this approach on top of the pseudoforest partition, one could potentially ruin the invariant that every vertex has out-degree no more than 1 in each pseudoforest, causing the algorithm of Henzinger et al. [27] to fail. We tackle this problem in steps. First, we show that if we were somehow able to invert the orientations of cycles, then we can make the moves of Blumenstock & Fischer's approach *faithful* to the degree condition of the pseudoforest algorithm of Henzinger et al. [27]. If we invert orientations along cycles in the pseudoforests, the out-degree of no vertex in the pseudoforests is changed. However, if we wish to perform these operations, we will have to do it in a manner that still allows us to maintain the underlying α'-bounded out-degree orientation. If the cycles are doubly η-valid, we invert the

cycles using Lemma 24. We do as in Section 3, but this time we add or subtract $1-\delta$ along the cycles. This ensures that every edge on the cycle now prefers the other endpoint, and so is naively rounded to the opposite direction without ending in H. The problem is that we have no guarantee that all edges are doubly η-valid. If an edge is only singly η-valid, then redistributing the load along a cycle containing this edge causes the edge to become invalid. However, by Lemma 11, we can delete such invalid edges and reinsert them again to restore the invariant that all edges are η-valid. We use a potential based argument to show that we can afford to perform these operations.

4.1 Ideas of Henzinger et al. and Blumenstock & Fischer

An α'-bounded out-degree orientation, can be split into α' pseudoforests by partitioning the edges such that each vertex has out-degree at most one in each partition. Then every connected component P_C in a partition is a pseudoforest. Indeed, $|E(P_C)| \leq |P_C|$ since every vertex has out-degree at most one. Hence, there can be at most one cycle in P_C. This idea is implicit in [27] by Henzinger et al. Note that we can store each pseudotree as a top tree with one extra edge with only $O(\log n)$ overhead per operation.

▶ **Lemma 27** (Implicit in [27]). *Given black box access to an algorithm maintaining an α'-bounded out-degree with update time $T(n)$, there exist an algorithm maintaining an α' pseudoforest decomposition with update time $O(T(n))$.*

Using the ideas of Blumenstock & Fischer [12], we can represent a pseudoforest P by a pair (F, M) s.t. F is a forest and M is matching, by adding exactly one edge from each cycle in P to M. Similarly, we can represent a partition of $E(G)$ into pseudoforests $(P_1, \ldots, P_k)$ by a pair (F, M) s.t. $F = \cup F_i$ and $M = \cup M_i$ and (F_i, M_i) represents P_i for all i.

In order to ensure the guarantees of Lemma 27, we need to maintain the invariant that every vertex has out-degree at most one in every pseudoforest. If this is the case, we say that the partition is *faithful* to the underlying orientation. Blumenstock & Fischer [12] perform operations on $G[M]$ in order to turn it into a forest. They call $G[M]$ the *surplus graph*. Some of the operations they perform, are described in the following lemma:

▶ **Lemma 28** (Implicit in [12]). *Let (F, M) be a faithful representation of a pseudoforest partition of a simple graph G equipped with an α'-bounded out-degree orientation. If $uv \in M_i$ and $vw \in M_j$ with $i \neq j$, then there exists an α'-bounded out-degree orientation with respect to which the partition gained by swapping $P_i \leftarrow P_i \cup \{vw\} - \{uv\}$ and $P_j \leftarrow P_j \cup \{uv\} - \{vw\}$ yields a faithful partition, and $uv \in M_j$ resp. $vw \in M_i$ iff. uv resp. vw are on the uni-cycle in their new pseudoforests.*

Furthermore, if $wx \in M_i$ for some x, then vw is not on a uni-cycle in P_i.

Proof. See [12] Lemma 2. To modify the orientation to accommodate the swaps, note that we can always reverse the direction of at most two cycles, without changing the out-degree of any vertex, such that both of the edges swapped are out-edges of v. Now swapping the two out-edges ensures that the partition stays faithful to the orientation (see Figure 2). ◀

Following Blumenstock & Fischer we note that if $e_1, \ldots, e_k$ is a path in a surplus graph $G[M]$ such that e_1 and e_k belong to the same matching M_i, then we can use the moves from Lemma 28 to restore colourfulness (see [12] Lemma 3). The key is that we can move the other edge in M_i towards e_1, and then after $O(k)$ switches, we are sure to end up in the furthermore part of Lemma 28. If a surplus graph contains no such paths, Blumenstock & Fischer say it is a *colourful* surplus graph. They show the following Lemma:

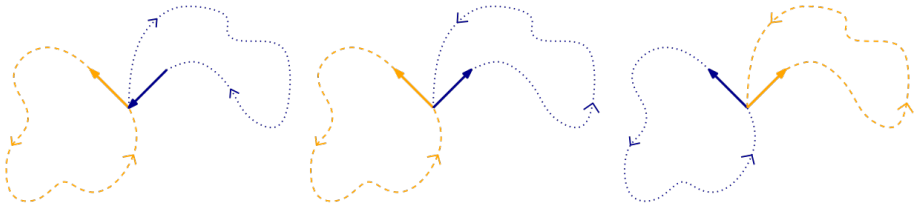

Figure 2 Moving edges between pseudoforests (represented by colour). Before performing the swap, we reorient a cycle so that the swapped edges are out-edges of their common endpoint. This ensures that every vertex has out-degree at most one in each pseudoforest.

▶ **Lemma 29** ([12]). *Suppose J is a colourful component of the surplus graph $G[M]$ of a graph G. Then for all $v \in J$ there exists an index i s.t. $N_{F_i}(v) \cap J = \emptyset$ and $J \cap M_i \neq \emptyset$.*

These Lemmas motivate the following approach: use Lemma 28 to ensure that the surplus graph is always colourful. Next use Lemma 28 to remove any cycles from the surplus graph.

4.2 Our algorithm for maintaining dynamic arboricity decompositions

Assume that we have an upper bound α_{max} on the arboricity throughout the entire update sequence. The algorithm works roughly as follows:

1. Run the algorithm from Theorem 26.
2. Naively round the orientation of each edge in $G - H$.
3. Split the rounded out-degree orientation on $G - H$ into pseudoforests.
4. Whenever an edge enters or moves between pseudoforests, we push it to a queue R.

We process each edge in $e \in R$ as follows: Put e into a pseudoforest. If e completes a cycle in a pseudoforest add it to $G[M]$. When e enters $G[M]$, we determine if it sits in a colourful component. If it doesn't, we apply Lemma 28 until all components in $G[M]$ are colourful. If a non-doubly valid cycle is reoriented in this process, we remove the singly-valid edges from the pseudoforests and add them to R. This ensures that the two edges from the same matching that we were trying to separate into two different components, are indeed separated. We will later bound the total number of edges pushed to R. If, on the other hand, the component is colourful, e may sit in a cyclic component. Then we apply Lemma 29 to remove the unique cycle. This may create a new non-colourful component, which we handle as before.

In the following, we describe the necessary data structures and sub-routines needed to perform these operations.

4.3 Operations on the surplus graph

In this section, we assume all cycles are doubly η-valid. In Section 4.5, we handle cases where this is not the case. Assuming that $G[M]$ is both colourful and acyclic, we can insert an edge in $G[M]$ and restore these invariants by performing switches according to Lemmas 28 and 29. Indeed, after inserting an edge, we can run, for example, a DFS on the component in $G[M]$ to determine if it is colourful. If it is not, we locate a path $e_1, \ldots, e_k$ such that $e_1, e_k \in M_i$. Then we apply Lemma 28 to e_{i-1} and e_i beginning with $i = k$, until an edge from M_i is removed from $G[M]$. Note that this is certain to happen when e_1 and e_3 belong to the same pseudoforest. We continue locating and handling paths until the component becomes colourful. If the component is colourful, but not acyclic, we choose a vertex v on the cycle and apply Lemma 29 to determine a pseudoforest represented in the component in which v is connected to no other vertex in the cyclic component. Then we determine a path

in the surplus graph between an edge in said pseudoforest and v. Now we move the edge in this pseudoforest to v using Lemma 28. If the edge is removed from $G[M]$ or the path is disconnected, we repeat the process. When such an edge is incident to v, we switch it with an edge on the cycle. Finally, we replace it in M with the unique neighbouring edge that is also incident to v in the cycle that put it in M. Now $G[M]$ is acyclic, but it may not be colourful. If this it the case, we repeat the arguments above until it becomes colourful. Note that these moves never create a cycle.

▶ **Lemma 30.** *After inserting an edge into $G[M]$, we can restore acyclicity and colourfulness in $O(\alpha^3 \log^2(n))$ time.*

4.4 Recovering neighbours

For each vertex, we will lazily maintain which of its out-edges belong to which pseudoforest. This costs only $O(1)$ overhead, when actually moving said edges. However, whenever we invert a cycle, these edges may change. Since the cycles can be long, we can only afford to update this information lazily, whenever the insertion/deletion algorithm determines the new out-neighbours of a vertex. When this happens, we say the vertex is *accessed*. Whenever an edge has its fractional load changed via a cycle inversion, it is always changed by the same amount. Hence, we make the following observation:

▶ **Observation 31.** *Between two accesses of a vertex v, the only possible new in-neighbours are the edges which were out-neighbours at the last access of v, and the only new out-neighbours are the vertices that are out-neighbours at the current access of v.*

Thus, we can recover exactly which incident edges might have changed in- and/or out-neighbour status from v, since the last time v was accessed by the insertion/deletion algorithm. To do so, we maintain that each top tree is rooted in the unique vertex, which has out-degree 0, when the underlying orientation is restricted to the tree. This ensures that we can recover v's unique out-edge in a pseudoforest by finding the first edge on the unique v-to-root path in the top tree. We maintain this information as follows:

- When we $link(u, v)$ with an edge oriented $u \to v$, we set the root of the new tree to be that of the tree containing v.
- When we $cut(u, v)$ with an edge oriented $u \to v$, we set the root of the tree containing u to be u and that containing v to be the same as the old tree.
- When we invert the orientation along a cycle originally oriented $u \to v \to \cdots \to u$, we change the root from v to u.
- When we perform a Lemma 29 swap, we also update the root accordingly.

Note that each update is accompanied by an operation costing $O(\log n)$ time, so the overhead for maintaining this information is only $O(1)$. With this information, we can recover the old out-neighbours as the stored out-neighbours, and the new out-neighbours by taking the first edge on the path from v to the root. Hence, we have shown:

▶ **Lemma 32.** *We can supply each vertex with a query returning a list L of neighbours which might have their status changed in time $O(\alpha \log n)$. Furthermore, $|L| = O(\alpha)$.*

4.5 Non doubly η-valid cycles

If a cycle is not doubly η-invalid, we still switch the orientation as before, but now we have to fix invalid edges. Assuming we know which edges have become η-invalidated, we fix them as follows: For every invalid edge, we first remove the edge from the pseudoforest it resides in.

This has two consequences. Firstly, the algorithm from Lemma 27 might move $O(1)$ edges between pseudoforests, and secondly, we also have to move an edge from the surplus graph back down as a normal edge in the pseudoforest it comes from. All of the (re)moved edges are pushed to the queue R. Then, we delete all invalid copies of edges in G^γ, and reinsert them. Now, all edges are valid again, and so we continue processing edges in R as described in Section 4.2. If an edge now belongs to H, we do not insert it into any pseudoforest.

It is important to note that the second consequence i.e. that we remove an edge from $G[M]$, either makes a Lemma 28 switch successful by removing one of the edges from M_i, or it removes an edge on one of the at most two paths between edges in M_i. In this case, we try to locate a second path, and handle it as before. This happens at most once: the component has at most one cycle, and hence at most two paths between two vertices. We ascribe the cost of deleting and reinserting invalid edges to the potential in Lemma 33 that bounds the total number of copies of edges that are inserted into G^γ. This cost is not ascribed to the algorithm maintaining $G[M]$. Set $\Delta^+_{max} = (1+\varepsilon)\alpha_{max}\gamma + \log_{(1+\varepsilon)} n$, we have:

▶ **Lemma 33.** *The total amount of insertions and deletions performed by the insertion / deletion algorithm over the entire update sequence is in* $O(\frac{\gamma\Delta^+_{max}}{\eta}(\Delta^+_{max} \cdot i + d))$

Lemma 33 allows us to bound the total number of edges moved between pseudoforests:

▶ **Lemma 34.** *We move at most* $O(\frac{\gamma(\Delta^+_{max})^3}{\eta}(i+d))$ *edges between pseudoforests.*

Note that this implies that the total no. of insertions into R is $O(\frac{\gamma(\Delta^+_{max})^3}{\eta}(i+d))$.

4.6 Locating singly η-valid edges

When we are accessing an edge, we can check if it is doubly η-valid or not (this information depends only on the load on the endpoints), and maintain this information in a dynamic forest using just 1 bit of information per edge. This allows us to later locate these edges using non-local searches in top trees. However, when edges go between being singly η-valid and doubly η-valid through operations not accessing said edge, we are not able to maintain this information. This can happen in two ways: either 1) a vertex has its load lowered causing an in-going edge to now become doubly η-valid or an outgoing edge to become singly η-valid or 2) a vertex has its load increased causing similar issues. We say an edge is *clean* if we updated the validity bit of an edge, the last time the out-degree of an endpoint of the edge was altered. Otherwise, we say it is *dirty*. Now if all edges on a cycle are clean, we can use top trees to direct searches for the edges that become invalidated by inverting the cycle.

We maintain a heavy-light decomposition of every forest using dynamic st-trees [40] to help us ensure that we can clean all edges in a cycle in time $O(\log^2 n)$. The idea is to maintain the invariant that all heavy edges are clean. Now we can clean a cycle by cleaning the at most $O(\log n)$ light edges on said cycle. In order to realise this invariant, whenever the degree of a vertex is changed, we need to update all of its incident heavy edges in all of the heavy-light decompositions. Since a vertex is incident to at most two heavy edges in each forest, we have to update $O(\alpha)$ heavy edges. The following holds:

▶ **Observation 35.** *We can locate singly η-valid edges on a clean cycle in time $O(\log n)$ per edge, if we spend $O(\log n)$ overhead updating the bit indicating double validity.*

▶ **Observation 36.** *We can insert and delete edges in the heavy-light decomposition in worst case $O(\log^2 n)$ time.*

▶ **Lemma 37.** *We can check if a cycle is doubly valid in time $O(\log^2 n)$.*

4.7 Conclusion

Theorem 3 and Corollary 4 follow from Lemma 38 (see full version for details).

▶ **Lemma 38.** *Consider a sequence of updates with i insertions and d deletions.*

1. *The insertion/deletion algorithm spends $O(\log^6(n) \cdot \alpha_{max}^4 \cdot \varepsilon^{-12}(i+d))$ time to update the fractional out-degree orientation and the refinement.*
2. *The algorithm maintaining the pseudoforests spends $O(\log^6(n) \cdot \alpha_{max}^3 \cdot \varepsilon^{-8}(i+d))$ time.*
3. *The algorithm maintaining the surplus graph spends $O(\log^6(n) \cdot \alpha_{max}^6 \cdot \varepsilon^{-8}(i+d))$ time.*

We have shown how to maintain an $\alpha+2$ arboricity decomposition of a fully dynamic graph as it undergoes an arboricity α preserving sequence of updates in $\text{poly}(\log n, \alpha)$ time per update. We have also shown how to maintain an $\lfloor(1+\varepsilon)\alpha\rfloor+2$ out-orientation of a fully dynamic graph in $\text{poly}(\log n, \alpha)$ time per update. These algorithms are the first dynamic algorithms to go below 2α forests and out-edges, respectively, and the number of forests matches the best near-linear static algorithm by Blumenstock and Fischer [12]. We apply these algorithms to get new trade-offs for implicit colouring algorithms for bounded arboricity graphs. In particular, we maintain $4 \cdot 2^\alpha$ and $2 \cdot 3^\alpha$ implicit colourings in $\text{poly}(\log n, \alpha)$ time per update. This improves upon the $2^{40\alpha}$ colours of the previous most colour-efficient algorithm maintaining $\text{poly}(\log n, \alpha)$ update time [27]. In particular, this reduces the number of colours for planar graphs from 2^{120} to 32. An interesting direction for future work is to see, if one can reduce the number of forests even further in the static case, while still achieving near-linear running time. Also, even though our algorithms use few colours and forests, the update times contain quite high polynomials in both $\log n$ and α. Is it possible to get more efficient update times without using more forests? Finally, for constant α, we get $\alpha+2$ out-edges. Brodal & Fagerberg [13] showed that one cannot get α out-edges with faster than $\Omega(n)$ update time (even amortised). The question remains, can one get $\alpha+1$?

5 Acyclic Orientations and Arboricity Decompositions

In this section, we briefly sketch the algorithm in Theorem 5 (Note that this section is partially based on the master's thesis by Christiansen [15, Chapter 9], see the full version for proofs). We modify an algorithm by Brodal & Fagerberg [13]. Specifically, we change how an edge is inserted. The algorithm maintains a list of out-edges $\text{out}(u)$ for each vertex $u \in G$. An edge e is in $\text{out}(u)$ if and only if $e \in E(G)$ and e is oriented away from u. As a result $d^+(u) = |\text{out}(u)|$. All of these lists are initialized to be empty. The algorithm ensures that the maximum out-degree of the vertices in G is d for some constant $d > 2\alpha$ to be specified later. The algorithm handles deletions and insertions in the following way:

Deletion: If e incident to x, y is deleted, we search $\text{out}(x)$ and $\text{out}(y)$ for e, and delete it.

Insertion: When an edge e is inserted, an arbitrary endpoint u of e is chosen, and e is added to $\text{out}(u)$. Now every edge in $\text{out}(u)$ is oriented in the other direction (also e) i.e. we delete $f = (u, v)$ from $\text{out}(u)$ and add f to $\text{out}(v)$ instead for all edges $f \in \text{out}(u)$. Now u has out-degree at most d, but the reorientations of an edge $f = (u, v)$ might increase $|\text{out}(v)|$ above d. The algorithm then proceeds by reorienting all out-edges out of v. It continues this process until all vertices have out-degree at most d.

Note that this process terminates: Since G has arboricity α, it has an orientation O such that the maximum out-degree in G is α. Call an edge in $E(G)$ *good*, if it is oriented the same way by both the algorithm and O and *bad* if it isn't. Now, inserting e could, in the worst case, make the algorithm change the orientation of α good edges. However from here on,

every vertex whose edges are reoriented will increase the total number of good edges by at least $d - 2\alpha \geq 1$, so the process terminates. The algorithm differs from the one presented in [13] in only one way. When an edge $e = uv$ is inserted, we always turn u into a sink. In [13], this only happens if u's out-degree increases above d. This small modification ensures no cycles are created: when an edge is inserted, one of its endpoints is turned into a sink, and so this edge is in no cycle. Also, turning a vertex into a sink does not create any cycles.

References

1 Oswin Aichholzer, Franz Aurenhammer, and Günter Rote. *Optimal graph orientation with storage applications*. SFB-Report , SFB 'Optimierung und Kontrolle'. TU Graz, 1995. Reportnr.: F003-51.

2 Stephen Alstrup, Jacob Holm, Kristian De Lichtenberg, and Mikkel Thorup. Maintaining information in fully dynamic trees with top trees. *ACM Trans. Algorithms*, 1(2):243–264, October 2005. doi:10.1145/1103963.1103966.

3 K. Appel and W. Haken. A proof of the four color theorem. *Discret. Math.*, 16(2):179–180, 1976. doi:10.1016/0012-365X(76)90147-3.

4 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. doi:10.1016/S0166-218X(97)00007-3.

5 Niranka Banerjee, Venkatesh Raman, and Saket Saurabh. Fully dynamic arboricity maintenance. In *Computing and Combinatorics - 25th International Conference, COCOON 2019, Xi'an, China, July 29-31, 2019, Proceedings*, volume 11653 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 2019. doi:10.1007/978-3-030-26176-4_1.

6 L. Barba, J. Cardinal, M. Korman, S. Langerman, A. van Renssen, M. Roeloffzen, and S. Verdonschot. Dynamic graph coloring. *Algorithmica*, 81(4):1319–1341, 2019.

7 Leonid Barenboim and Michael Elkin. Sublogarithmic distributed mis algorithm for sparse graphs using nash-williams decomposition. In *Proceedings of the Twenty-Seventh ACM Symposium on Principles of Distributed Computing*, PODC '08, pages 25–34, New York, NY, USA, 2008. Association for Computing Machinery. doi:10.1145/1400751.1400757.

8 Edvin Berglin and Gerth Stolting 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. doi:10.4230/LIPIcs.ISAAC.2017.12.

9 Sayan Bhattacharya, Deeparnab Chakrabarty, Monika Henzinger, and Danupon Nanongkai. Dynamic algorithms for graph coloring. In *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. doi:10.1137/1.9781611975031.1.

10 Sayan Bhattacharya, Fabrizio Grandoni, Janardhan Kulkarni, Quanquan C. Liu, and Shay Solomon. Fully dynamic (Δ+1)-coloring in constant update time. *CoRR*, abs/1910.02063, 2019. arXiv:1910.02063.

11 Markus Blumenstock. Fast algorithms for pseudoarboricity. In *Proceedings of the Eighteenth Workshop on Algorithm Engineering and Experiments, ALENEX 2016, Arlington, Virginia, USA, January 10, 2016*, pages 113–126. SIAM, 2016. doi:10.1137/1.9781611974317.10.

12 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. doi:10.1007/978-3-030-38919-2_5.

13 Gerth Stolting 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.

14 Norishige Chiba, Takao Nishizeki, and Nobuji Saito. A linear 5-coloring algorithm of planar graphs. *J. Algorithms*, 2(4):317–327, 1981. doi:10.1016/0196-6774(81)90031-6.

15 Aleksander B. G. Christiansen. Dynamic algorithms for implicit vertex-colouring of graphs with bounded arboricity. Master's thesis, Technical University of Denmark, Kgs. Lyngby, Denmark, October 2021.

16 Aleksander B. G. Christiansen and Eva Rotenberg. Fully-dynamic α+2 arboricity decomposition and implicit colouring. *CoRR*, abs/2203.06039, 2022. doi:10.48550/arXiv.2203.06039.

17 O. Coudert. Exact coloring of real-life graphs is easy. *Proceedings of 34th Design Automation Conference. ACM*, 35(1):121–126, 1997.

18 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.

19 David Eppstein. Arboricity and bipartite subgraph listing algorithms. *Inf. Process. Lett.*, 51(4):207–211, August 1994. doi:10.1016/0020-0190(94)90121-X.

20 Greg N. Frederickson. On linear-time algorithms for five-coloring planar graphs. *Inf. Process. Lett.*, 19(5):219–224, 1984. doi:10.1016/0020-0190(84)90056-5.

21 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, pages 407–421, New York, NY, USA, 1988. Association for Computing Machinery. doi:10.1145/62212.62252.

22 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, pages 88–97, USA, 1995. Society for Industrial and Applied Mathematics.

23 Mohsen Ghaffari and Hsin-Hao Su. Distributed degree splitting, edge coloring, and orientations. 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 2505–2523. SIAM, 2017. doi:10.1137/1.9781611974782.166.

24 David G. Harris. Distributed local approximation algorithms for maximum matching in graphs and hypergraphs. In *2019 IEEE 60th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 700–724, 2019. doi:10.1109/FOCS.2019.00048.

25 David G. Harris, Hsin-Hao Su, and Hoa T. Vu. On the locality of nash-williams forest decomposition and star-forest decomposition. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, PODC'21, pages 295–305, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3465084.3467908.

26 Meng He, Ganggui Tang, and N. Zeh. Orienting dynamic graphs, with applications to maximal matchings and adjacency queries. In *ISAAC*, 2014.

27 Monika Henzinger, Stefan Neumann, and Andreas Wiese. Explicit and implicit dynamic coloring of graphs with bounded arboricity. *CoRR*, abs/2002.10142, 2020. arXiv:2002.10142.

28 Monika Henzinger and Pan Peng. Constant-time dynamic (Δ+1)-coloring. In *37th International Symposium on Theoretical Aspects of Computer Science, STACS 2020, March 10-13, 2020, Montpellier, France*, volume 154 of *LIPIcs*, pages 53:1–53:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.STACS.2020.53.

29 Subhash Khot and Ashok Kumar Ponnuswami. Better inapproximability results for maxclique, chromatic number and min-3lin-deletion. In *Automata, Languages and Programming, 33rd International Colloquium, ICALP 2006, Venice, Italy, July 10-14, 2006, Proceedings, Part I*, volume 4051 of *Lecture Notes in Computer Science*, pages 226–237. Springer, 2006. doi:10.1007/11786986_21.

30 Tsvi Kopelowitz, Robert Krauthgamer, Ely Porat, and Shay Solomon. Orienting fully dynamic graphs with worst-case time bounds. In *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. doi:10.1007/978-3-662-43951-7_45.

31 Łukasz Kowalik. Approximation scheme for lowest outdegree orientation and graph density measures. In *Proceedings of the 17th International Conference on Algorithms and Computation*, ISAAC'06, pages 557–566, Berlin, Heidelberg, 2006. Springer-Verlag. doi:10.1007/11940128_56.

32 Łukasz Kowalik. Adjacency queries in dynamic sparse graphs. *Inf. Process. Lett.*, 102(5):191–195, May 2007. doi:10.1016/j.ipl.2006.12.006.

33 Yin Tat Lee and Aaron Sidford. Path finding methods for linear programming: Solving linear programs in $\tilde{O}$(vrank) iterations and faster algorithms for maximum flow. In *2014 IEEE 55th Annual Symposium on Foundations of Computer Science*, pages 424–433, 2014. doi:10.1109/FOCS.2014.52.

34 Aleksander Madry. Navigating central path with electrical flows: From flows to matchings, and back. In *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*, pages 253–262, 2013. doi:10.1109/FOCS.2013.35.

35 D. Matula, Y. Shiloach, and R. Tarjan. Two linear-time algorithms for five-coloring a planar graph, 1980. Reportnr.: STAN-CS-80-830.

36 C. St.J. A. Nash-Williams. Decomposition of finite graphs into forests. *Journal of the London Mathematical Society*, s1-39(1):12–12, 1964. doi:10.1112/jlms/s1-39.1.12.

37 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. doi:10.1002/net.3230120206.

38 Neil Robertson, Daniel P. Sanders, Paul D. Seymour, and Robin Thomas. Efficiently four-coloring planar graphs. In Gary L. Miller, editor, *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, Philadelphia, Pennsylvania, USA, May 22-24, 1996*, pages 571–575. ACM, 1996. doi:10.1145/237814.238005.

39 Saurabh Sawlani and Junxing Wang. Near-optimal fully dynamic densest subgraph. In *Proccedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing, STOC 2020, Chicago, IL, USA, June 22-26, 2020*, pages 181–193. ACM, 2020. doi:10.1145/3357713.3384327.

40 Daniel D. Sleator and Robert Endre Tarjan. A data structure for dynamic trees. In *Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing*, STOC '81, pages 114–122, New York, NY, USA, 1981. Association for Computing Machinery. doi:10.1145/800076.802464.

41 Shay Solomon and Nicole Wein. Improved dynamic graph coloring. *ACM Trans. Algorithms*, 16(3), June 2020. doi:10.1145/3392724.

42 Hsin-Hao Su and Hoa T. Vu. Distributed Dense Subgraph Detection and Low Outdegree Orientation. In Hagit Attiya, editor, *34th International Symposium on Distributed Computing (DISC 2020)*, volume 179 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 15:1–15:18, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.DISC.2020.15.

43 David Zuckerman. Linear degree extractors and the inapproximability of max clique and chromatic number. In *Proceedings of the Thirty-Eighth Annual ACM Symposium on Theory of Computing*, STOC '06, pages 681–690, New York, NY, USA, 2006. Association for Computing Machinery. doi:10.1145/1132516.1132612.