Orienting Fully Dynamic Graphs with Worst-Case Time Bounds[⋆]

Tsvi Kopelowitz[1,⋆⋆], Robert Krauthgamer[2,⋆⋆⋆], Ely Porat[3], and Shay Solomon[4,†]

[1] University of Michigan, USA
kopelot@gmail.com
[2] Weizmann Institute of Science, Israel
robert.krauthgamer@weizmann.ac.il
[3] Bar-Ilan University, Israel
poratel y@cs.biu.ac.il
[4] Weizmann Institute of Science, Israel
shay.solomon@weizmann.ac.il

Abstract. In edge orientations, the goal is usually to orient (direct) the edges of an undirected network (modeled by a graph) such that all out-degrees are bounded. When the network is fully dynamic, i.e., admits edge insertions and deletions, we wish to maintain such an orientation while keeping a tab on the update time. Low out-degree orientations turned out to be a surprisingly useful tool for managing networks.

Brodal and Fagerberg (1999) initiated the study of the edge orientation problem in terms of the graph's arboricity, which is very natural in this context. Their solution achieves a constant out-degree and a logarithmic *amortized* update time for all graphs with constant arboricity, which include all planar and excluded-minor graphs. It remained an open question – first proposed by Brodal and Fagerberg, later by Erickson and others – to obtain similar bounds with *worst-case* update time.

We address this 15 year old question by providing a simple algorithm with worst-case bounds that nearly match the previous amortized bounds. Our algorithm is based on a new approach of maintaining a combinatorial invariant, and achieves a logarithmic out-degree with logarithmic worst-case update times. This result has applications to various dynamic network problems such as maintaining a maximal matching, where we obtain logarithmic worst-case update time compared to a similar amortized update time of Neiman and Solomon (2013).

1 Introduction

A very useful algorithmic tool for managing networks is to *orient* (direct) the edges while providing a guaranteed upper bound on the out-degree of every vertex.

[⋆] A full version appears at http://arxiv.org/abs/1312.1382

[⋆⋆] This work is supported by NSF grants CCF-1217338 and CNS-1318294.

[⋆⋆⋆] Work supported in part by a US-Israel BSF grant #2010418, an Israel Science Foundation grant #897/13, and by the Citi Foundation.

[†] This work is supported by the Koshland Center for basic Research.

Formally, an orientation of an undirected graph $G = (V, E)$ is called a *c-orientation* if every vertex has out-degree at most $c \geq 1$.

There are many examples where orientations are used in the design, maintenance and manipulation of networks, including both static and dynamic networks, and both centralized algorithms and distributed ones. One exciting example of the power of graph orientations can be seen in the seminal paper introducing "color-coding" [1], where orientations are used to develop more efficient algorithms for finding simple cycles and paths. Another fundamental example is in data structures for quickly answering adjacency queries [2,3,4], where a c-orientation of a (dynamic) graph G is used to answer adjacency queries in $O(c)$ time using only linear space. These techniques [2,3,4] were further generalized to answer short-path queries [5]. Additional examples for the algorithmic use of low-degree orientations include load balancing [6], maximal matchings [7], counting subgraphs in sparse graphs [8], prize-collecting TSPs and Steiner Trees [9], reporting all maximal independent sets [10], answering dominance queries [10], subgraph listing problems (listing triangles and 4-cliques) in planar graphs [2], and computing the girth [5].

Efficient Data Communication. The efficiency of network communication can often be improved significantly by assigning one endpoint of every edge as "responsible" for all data transfers occurring on that edge. Such a responsibility assignment can be naturally obtained by orienting the graph's edges and letting each vertex be responsible only for its outgoing edges. Consider, for example, the task of computing some aggregate function of dynamic data that resides locally at a vertex and its neighbors – can this task be carried out without scanning all the neighbors of that vertex? Given a c-orientation, whenever the local data in a vertex u changes, u updates all its outgoing neighbors (neighbors of u through edges oriented out of u). In contrast, u need not update any of its (possibly many) incoming neighbors (neighbors of u through edges oriented into u) about this change. When u wishes to compute the function, it only needs to scan its outgoing neighbors in order to gather the full up-to-date data. Such responsibility assignment is particularly useful in dynamic networks, see [7] for an example, and could be very effective also in many standard tasks in distributed, self-stabilizing, peer-to-peer, or ad-hoc networks, such as reducing the message complexity, or reducing local memory constraints (e.g., a router would only store information about its c outgoing neighbors).

Dynamic Graphs. Our focus here is on maintaining low out-degree orientations of fully dynamic graphs on n fixed vertices, where edge updates (insertions and deletions) take place over time. The goal is to develop efficient and simple algorithms that guarantee that the maximum out-degree in the (dynamic) orientation of the graph is small. In particular, we are interested in obtaining non-trivial update times that hold (1) in the *worst-case*, and (2) *deterministically.* Notice that in order for an update algorithm to be efficient, the number of *edge re-orientations* (done when performing an edge update) must be small, as this number is clearly a lower bound for the algorithm's update time.

The out-degree bound achieved by our algorithms will be expressed in terms of the sparsity of the graph, as measured by the *arboricity* of G (defined below), which is a natural lower bound for the maximum out-degree of any orientation.

Arboricity. The *arboricity* of an undirected graph $G = (V, E)$ is defined as $\alpha(G) = \max_{U \subseteq V} \left\lceil \frac{|E(U)|}{|U|-1} \right\rceil$, where $E(U)$ is the set of edges induced by U (which we assume has size $|U| \geq 2$). This is a concrete formalism for the notion of everywhere-sparse graphs — every subgraph of G has arboricity at most $\alpha(G)$ as well. Arboricity and its related sparseness measures of *thickness, degeneracy* or *density*, which are all equal up to a constant factor, were studied extensively. Most notable in this context is the family of graphs with constant arboricity, which includes all excluded-minor graphs, and in particular planar graphs and bounded-treewidth graphs.

A key property of bounded arboricity graphs that has been exploited in various algorithmic applications is the following Nash-Williams Theorem.

Theorem 1 (Nash-Williams [11,12]). *A graph $G = (V, E)$ has arboricity $\alpha(G)$ if and only if $\alpha(G) > 0$ is the smallest number of sets $E_1, \ldots, E_{\alpha(G)}$ that E can be partitioned into, such that each subgraph (V, E_i) is a forest.*

This theorem implies that the edges of an undirected graph $G = (V, E)$ can be oriented such that the out-degree of each vertex is at most $\alpha(G)$. To see this, consider the guaranteed partition $E_1, \ldots, E_{\alpha(G)}$. For each forest (V, E_i) and each tree in that forest, designate one arbitrary vertex as the root of that tree, and orient all edges towards that root. In each oriented forest the out-degree of every vertex is at most 1, hence the union of the oriented forests has out-degree bounded by $\alpha(G)$. There exists a polynomial-time algorithm that computes for a (static) graph G the exact arboricity $\alpha(G)$ [13], and a linear-time algorithm that computes a $(2\alpha(G) - 1)$-orientation [14].

For every (static) graph G, the minimum-possible maximum out-degree is closely related to $\alpha(G)$: the argument above provides an orientation with maximum out-degree at most $\alpha(G)$, but the maximum out-degree is also easily seen to be at least $\alpha(G) - 1$ (for every orientation).[1] In other words, the arboricity measure of sparsity is a natural baseline for low out-degree orientations.

1.1 Main Result

We obtain efficient algorithms for maintaining a low out-degree orientation of a fully dynamic graph G such that the out-degree of each vertex is small and the running time of all update operations is bounded in the worst-case. Specifically, we present two algorithms. The first algorithm achieves (at any point in time)

[1] To see this, let $U \subset V$ be such that $\left\lceil \frac{|E(U)|}{|U|-1} \right\rceil = \alpha(G)$, hence $\frac{|E(U)|}{|U|-1} > \alpha(G) - 1$. For every orientation, the maximum out-degree in G is at least the average out-degree of vertices in U, which in turn is at least $\frac{|E(U)|}{|U|} > \frac{|U|-1}{|U|}(\alpha(G) - 1)$. The bound now follows from both $\alpha(G)$ and the maximum out-degree being integers.

– a maximum out-degree $\Delta \leq \inf_{\beta>1}\{\beta \cdot \alpha(G) + \lceil \log_\beta n \rceil\}$, and
– insertion and deletion update times $O(\Delta^2)$ and $O(\Delta)$, respectively.

The second algorithm works with two parameters $\hat{\alpha}$ and $\hat{\beta} > 1$ both known by the algorithm. The parameter $\hat{\alpha}$ is a set upper bound on $\alpha(G)$ while $\hat{\beta}$ can be chosen arbitrarily and only affects the complexities of the algorithm. This algorithm achieves (at any point in time)

– a maximum out-degree $\Delta \leq \hat{\beta} \cdot \hat{\alpha} + \lceil \log_{\hat{\beta}} n \rceil$, and
– insertion and deletion update times $O(\hat{\beta} \cdot \hat{\alpha} \cdot \Delta)$ and $O(\Delta)$, respectively.

Notice that the first algorithm does not need to know $\alpha(G)$ (hence its bounds change with time together with the graph G), while the second algorithm assumes knowledge of an upper bound on $\alpha(G)$. On the other hand, the second algorithm has faster insertion time, because in the worst-case $\hat{\beta} \cdot \hat{\alpha} \cdot \Delta < \Delta^2$.

All our algorithms are deterministic, and they change the orientation of at most $\Delta + 1$ edges per edge update. Perhaps most importantly, they are relatively simple (especially the first one) to describe and to analyze, which is a great virtue for potential implementation, and also for further extensions and refinements. We should nevertheless point out that the apparent simplicity relies heavily on a fine selection of an effective combinatorial invariant; finding such invariants can be very tricky, and it constitutes the main technical challenge in this work.

Notice that in our second algorithm if $\hat{\alpha}$ is constant we can set $\hat{\beta} = 2$ and all of our bounds translate to $O(\log n)$. In other words, for fully dynamic graphs with a constant upper bound on the arboricity we can maintain an $O(\log n)$-orientation with $O(\log n)$ worst-case update time. Previous work, which is discussed next, only obtained efficient *amortized* update time bounds, in contrast to our bounds which are all in the worst-case. Our results address an open question raised by Brodal and Fagerberg [3] and restated by Erickson [15], of obtaining good worst-case bounds (although the ultimate goal is obviously worst-case time $O(1)$ for all updates, if that is at all possible).

1.2 Comparison with Previous Work

The dynamic setting in our context was pioneered by Brodal and Fagerberg [3], who showed that it is possible to maintain a $4\hat{\alpha}$-orientation of a fully dynamic graph G whose arboricity is always at most $\hat{\alpha}$. They proved that their algorithm is $O(1)$-competitive against the number of re-orientations made by any algorithm, regardless of that algorithm's actual running time. They then provided a specific strategy for re-orienting edges which shows that, for $\hat{\alpha} = O(1)$, their algorithm's insertion time is *amortized* $O(1)$ while the deletion time is *amortized* $O(\log n)$. Kowalik [4] showed that a different analysis of Brodal and Fagerberg's algorithm achieves insertion update time that is *amortized* $O(\log n)$ and the deletion time that is *worst-case* $O(1)$. Kowalik further showed it is possible to support insertions in *amortized* $O(1)$ time and deletions in worst-case $O(1)$ time by using an $O(\log n)$-orientation. These algorithms have been used as black-box components in several applications of dynamic graphs. Recently Gupta et.al. [16] showed that if only insertions are allowed then an amortized 2 edge reorientations suffice for maintaining a maximum out-degree of $O(\alpha(G))$.

Algorithms with amortized runtime bounds may be insufficient for many real-time applications where infrequent costly operations might cause congestion in the system at critical times. Exploring the boundaries between amortized and worst-case bounds is also important from a theoretical point of view, and has received a lot of research attention. The algorithms of Brodal and Fagerberg [3] and Kowalik [4] both incur a *linear worst-case update time*, on which we show an exponential improvement. As mentioned above, our results address an open question raised by Brodal and Fagerberg [3] and restated by Erickson in [15].

1.3 Our Techniques

The algorithm of Brodal and Fagerberg [3] is very elegant, but it is not clear if it can be deamortized as it is inherently amortized. The key technical idea we introduce is to maintain a combinatorial invariant, which is very simple in its basic form: for every vertex $u \in V$, at least (roughly) $\hat{\alpha}$ outgoing edges are directed towards vertices with almost as large out-degree, namely at least $d_{out}(u) - 1$ (where $d_{out}(u)$ is the out-degree of u). Such edges are called *valid* edges. We prove in Section 2 that this combinatorial invariant immediately implies the claimed upper bound on Δ.

An overview of the algorithms that we use for, say, insertion, is as follows. When a new edge (u, v) is added, we first orient it, say, from u to v guaranteeing that the edge is valid. We now check if the invariant holds, but the only culprit is u, whose out-degree has increased. If we know which of the edges leaving u are the "special" valid edges needed to maintain the invariant, we scan them to see if any of them are no longer valid (as a result of the insertion), and if there is such an edge we *flip* its orientation, and continue recursively with the other endpoint of the flipped edge. This process indeed works, but it causes difficulty during an edge deletion — when one of the $\hat{\alpha}$ special valid edges leaving u is deleted, a replacement may not even exist.

Here, our expedition splits into two different parts. We first show an extremely simple (but less efficient) algorithm that maintains a stronger invariant in which for every vertex $u \in V$, *all* of its out-going edges are valid. This approach immediately gives the claimed upper bound on Δ, with update time roughly $O((\frac{\log n}{\log \log n})^2)$ for graphs with constant arboricity.

In the second part we refine the invariant using another idea of *spectrum-validity*, which roughly speaking uses the following invariant: for every vertex $u \in V$ and for every $1 \le i \le \frac{\deg(u)}{\hat{\alpha}}$, at least $i \cdot \hat{\alpha}$ of its outgoing edges are directed towards vertices with degree at least $d_{out}(u) - i$. This invariant is stronger than the first invariant (which seemed algorithmically challenging) and weaker than the second invariant (whose bounds were less efficient than desired as it needed to guarantee validness for all edges). Furthermore, maintaining this invariant is more involved algorithmically, and one interesting aspect of our algorithm is that during an insertion process, it does not scan the roughly $\hat{\alpha}$ neighbors with degree at least $d_{out}(u) - 1$, as one would expect, but rather some other neighbors picked in a careful manner. Ultimately, this methodology yields the improved time bounds claim in Section 1.1.

1.4 Selected Applications

We only mention two applications here by stating their theorems for graphs with arboricity bounded by a constant. We discuss these applications and some other ones with more detail in the full version.

Theorem 2 (Maximal matching in fully dynamic graphs). *Let $G = (V, E)$ be an undirected fully dynamic graph with arboricity bounded by a constant. Then one can deterministically maintain a maximal matching of G such that the worst-case time per edge update is $O(\log n)$.*

Theorem 3 (Adjacency queries in fully dynamic graphs). *Let $G = (V, E)$ be an undirected fully dynamic graph with arboricity bounded by a constant. Then one can deterministically answer adjacency queries on G in $O(\log\log\log n)$ worst-case time where the deterministic worst-case time per edge update is $O(\log n \cdot \log\log\log n)$.*

1.5 Preliminaries

An *orientation* of the undirected edges of G assigns a direction to every edge $e \in E$, thereby turning G into a digraph. We will use the notation $u \to v$ to indicate that the edge $e = (u, v)$ is oriented from u to v. Given such an orientation, let $N^+(u) := \{v \in V : u \to v\}$ denote the set of *outgoing neighbors* of u, i.e., the vertices connected to u via an edge leaving it, and let $d_{out}(u) := |N^+(u)|$ denote the number of *outgoing edges* of u in this orientation, i.e., the *out-degree* of u. Similarly, let $N^-(u) := \{v \in V : v \to u\}$ denote the set of *incoming neighbors* of u, and let $d_{in}(u) := |N^-(u)|$. Finally, we denote by $\Delta := \max_{v\in V} d_{out}(v)$ the maximum out-degree of a vertex in the graph (under the given orientation).

Our algorithms will make use of the following heap-like data structure (the proof is left for the full version).

Lemma 1. *Let X be a dynamic set, where each element $x_i \in X$ has a key $k_i \in \mathbb{N}$ that may change with time, and designate a fixed element $x_0 \in X$ to be the* center *of X (although its key k_0 may change with time). Then there is a data structure that maintains X using $O(|X| + k_0)$ words of space, and supports the following operations with $O(1)$ worst-case time bound (unless specified otherwise):*

- ReportMax(X)*: return a pointer to an element from X with maximum key.*
- Increment(X, x)*: given a pointer to $x \in X \setminus \{x_0\}$, increment the key of x.*
- Decrement(X, x)*: given a pointer to $x \in X \setminus \{x_0\}$, decrement the key of x.*
- Insert(X, x_i, k_i)*: insert a new element x_i with key $k_i \leq k_0 + 1$ into X.*
- Delete(X, x)*: given a pointer to an element $x \in X \setminus \{x_0\}$, remove x from X.*
- IncrementCenter(X)*: increment k_0 in $O(k_0)$ worst-case time.*
- DecrementCenter(X)*: decrement k_0 (unless $k_0 = 1$) in $O(k_0)$ worst-case time.*

For each vertex $w \in V$, consider the (dynamic) set X_w that contains w and all its incoming neighbors, where the key of each element in X is given by its out-degree. The center element of X_w will be w itself. Each vertex w will have its own data structure (using Lemma 1) for maintaining X_w. In what follows, we denote this data structure by H_w, and use it to find an incoming neighbor of w with out-degree at least $d_{out}(w) + 2$ (if one exists) in $O(1)$ time.

Lemma 2. *The total space used to store the data structures H_w for all $w \in V$ is $O(n + m)$ words, where m stands for the number of edges in the (current) graph.*

Proof. By Lemma 1, for each $w \in V$ the space usage is at most $O(1 + d_{in}(w) + d_{out}(w))$. Summing over all vertices $w \in V$, the total space is $\sum_{w \in V} O(1 + d_{in}(w) + d_{out}(w)) = O(n + m)$. □

2 Invariants for Bounding the Largest Out-Degree

We assume throughout that the dynamic graph G has, at all times, arboricity $\alpha(G)$ bounded by some parameter $\hat{\alpha}$, i.e., $\alpha(G) \leq \hat{\alpha}$. Let $\hat{\beta} > 1$ be a parameter that may possibly depend on n and $\hat{\alpha}$ (it will be chosen later to optimize our bounds), and define $\gamma := \hat{\beta} \cdot \hat{\alpha}$.

An edge $(u, v) \in E$ oriented such that $u \to v$ is called *valid* if $d_{out}(u) \leq d_{out}(v) + 1$, and is called *violated* otherwise. The following condition provides control (upper bound) on Δ, as proved in Theorem 4. We refer to it as an *invariant*, because we shall maintain the orientation so that the condition is satisfied at all times.

Invariant 3. *For each vertex w, at least $\min\{d_{out}(w), \gamma\}$ outgoing edges of w are valid.*

Theorem 4. *If Invariant 3 holds, then $\Delta \leq \hat{\beta} \cdot \hat{\alpha} + \lceil \log_{\hat{\beta}} n \rceil$.*

Proof. Assume Invariant 3 holds, and suppose for contradiction there is a "source" vertex $s \in V$ satisfying $d_{out}(s) > \gamma + \lceil \log_{\hat{\beta}} n \rceil$. Now consider the set V_i of vertices reachable from s by directed paths of length at most i that use only valid edges. Observe that for every $1 \leq i \leq \lceil \log_{\hat{\beta}} n \rceil$ and every vertex $w \in V_i$,

$$d_{out}(w) \geq d_{out}(s) - i > \gamma + \lceil \log_{\hat{\beta}} n \rceil - i \geq \gamma,$$

implying that at least γ outgoing edges of w are valid.

We next prove by induction on i that $|V_i| > \hat{\beta}^i$ for all $1 \leq i \leq \lceil \log_{\hat{\beta}} n \rceil$. For the base case $i = 1$, notice that s has at least γ valid outgoing edges and all of the corresponding outgoing neighbors of s belong to V_1. Furthermore, s belongs to V_1 as well. Thus $|V_1| \geq \gamma + 1 > \gamma \geq \hat{\beta}$. For the inductive step, suppose $|V_{i-1}| > \hat{\beta}^{i-1}$; observe that the total number of valid outgoing edges from vertices in V_{i-1} is at least $\gamma |V_{i-1}|$, and furthermore all these edges are incident only to vertices in V_i. Since the graph's arboricity is $\alpha(G) \leq \hat{\alpha}$, we can bound $|V_i| - 1 \geq \gamma |V_{i-1}| / \alpha(G) \geq \hat{\beta} |V_{i-1}| > \hat{\beta}^i$, as claimed.

We conclude that $|V_{\lceil \log_{\hat{\beta}} n \rceil}| > \hat{\beta}^{\lceil \log_{\hat{\beta}} n \rceil} \geq n$, yielding a contradiction. □

Invariant 3 provides a relatively weak guarantee as if $d_{out}(w) > \gamma$, then we know only that γ outgoing edges of w are valid, and have no guarantee on the out-degree of the other $d_{out}(w) - \gamma$ outgoing neighbors of w. Consequently, it is nontrivial to maintain Invariant 3 efficiently, and in particular, if one of the γ valid edges (outgoing from w) is deleted, the invariant might become violated, and it is unclear how to restore it efficiently. We thus need another invariant, namely, a stronger condition (so that a similar theorem still applies) that is also easy to maintain. The next invariant is a natural candidate, as it is simple to maintain (with reasonable efficiency).

Invariant 4. *All edges in G are valid.*

Theorem 5. *If Invariant 4 holds, then $\Delta \leq \inf_{\beta>1} \beta \cdot \alpha(G) + \lceil \log_\beta n \rceil$.*

The proof of Theorem 5 is similar to the proof of Theorem 4 and is left for the full version.

We first present in Section 3 a very simple algorithm that maintains Invariant 4 with update times $O(\Delta^2)$ and $O(\Delta)$ for insertion and deletion (of an edge), respectively. This algorithm provides a strong basis for a more sophisticated algorithm, developed in Section 4, which maintains an intermediate invariant (stronger than Invariant 3 but weaker than Invariant 4) with update times $O(\gamma \cdot \Delta)$ and $O(\Delta)$ for insertion and deletion, respectively.

3 Worst-Case Algorithm

We consider an infinite sequence of graphs $G_0, G_1, \ldots$ on a fixed vertex set V, where each graph $G_i = (V, E_i)$ is obtained from the previous graph G_{i-1} by either adding or deleting a single edge. For simplicity, we assume that G_0 has no edges. Denote by $\alpha_i = \alpha(G_i)$ the arboricity of G_i. We will maintain Invariant 4 while edges are inserted and deleted into and from the graph, which by Theorem 5 implies that the maximum out-degree Δ_i in the orientation of G_i is bounded by $O(\inf_{\beta>1}\{\beta \cdot \alpha_i + \log_\beta n\})$.

For the rest of this section we fix i and consider a graph G_i obtained from a graph G_{i-1} satisfying Invariant 4 by either adding or deleting edge $e = (u, v)$.

3.1 Insertions

Suppose that edge (u, v) is added to G_{i-1} thereby obtaining G_i. We begin by orienting the edge from the endpoint with lower out-degree to the endpoint with larger out-degree (breaking a tie in an arbitrary manner). So without loss of generality we now have $u \to v$. Notice that the only edges that may be violated now are edges outgoing from u, as $d_{out}(u)$ is the only out-degree that has been incremented. Furthermore, if some edge $u \to v'$ is violated now, then removing this edge will guarantee that there are no violated edges. However, the resulting graph would be missing the edge (u, v') just removed. So we recursively insert the edge (u, v'), but orient it in the opposite direction (i.e., $v' \to u$). This means that we have actually *flipped* the orientation of (u, v'), reverting $d_{out}(u)$ to its value before the entire insertion process took place. This recursive process continues until all edges of the graph are valid. Moreover, at any given time there is at most one "missing" edge, and the graph obtained at the end of the process has no missing edges. Our choice to remove a violated edge outgoing from u (if such an edge exists) guarantees that the number of recursive steps is at most Δ, as we will show later. This insertion process is described in Algorithm 1.

Algorithm 1. Recursive-Insertion$(G, (u, v))$

```
/* Assume without loss of generality $d_{out}(u) \leq d_{out}(v)$ */
 1. add $(u, v)$ to $G$ with orientation $u \to v$
 2. Insert$(X_v, u, d_{out}(u) - 1)$ /* this key will be incremented in line 10 if needed */
 3. for $v' \in N^+(u)$ do
 4.    if $d_{out}(u) > d_{out}(v') + 1$ then
 5.       remove $(u, v')$ from $G$ /* now edge $(u, v')$ is missing */
 6.       Delete$(X_{v'}, u)$
 7.       Recursive-Insertion$(G, (v', u))$ /* recursively insert $(u, v')$, but oriented $v' \to u$
          */
 8.       return
 9. for $v' \in N^+(u)$ do
10.    Increment$(X_{v'}, u)$
11. IncrementCenter$(X_u)$
```

We remark that although in line 1 the out-degree of u is incremented by 1, we do not update the new key of u in the appropriate structures (i.e., H_u and $H_{v'}$ for all $v' \in N^+(u)$), because if the condition in line 4 succeeds for some $v' \in N^+(u)$, the out-degree of u will return to its original value, and we want to save the cost of incrementing and then decrementing the key for u in all structures. However, if that condition fails for all v', we will perform the update in lines 9–11.

Correctness and Runtime Analysis. The following lemmas, provide the correctness and runtime analysis of the insertion process. Due to space constraints, the proofs are omitted here and appear in the full version. Notice that the proofs mostly follow from the discussion above.

Lemma 5. *At the end of the execution of* Recursive-Insertion *on an input graph which has an orientation satisfying Invariant 4, Invariant 4 holds for the resulting graph and orientation.*

Lemma 6. *The total number of recursive calls (and hence re-orientations) of* Recursive-Insertion *due to an insertion into G is at most $\Delta + 1$, and the total runtime is bounded by $O(\Delta^2)$.*

3.2 Deletions

Suppose that edge (u, v) is deleted from G_{i-1} thereby obtaining G_i. Assume without loss of generality that in the orientation of G_{i-1} we had $u \to v$.

We begin by removing (u, v) from our data structure. Notice that the only edges that may be violated now are edges incoming into u. Furthermore, if there is an edge $v' \to u$ that is violated now, then adding to the graph another copy of (u, v') (producing a multi-graph) that is oriented in the opposite direction (i.e., $u \to v'$) will guarantee that there are no violated edges. However, the resulting multi-graph has an extra edge that should be deleted. So we now recursively delete the original copy of edge (u, v') (not the copy that was just added, oriented $u \to v'$, which we keep). This means that we have actually *flipped* the orientation of (u, v'), reverting $d_{out}(u)$ to its value before the entire deletion process took place. This recursive process will continue until all edges of the graph are valid. Moreover, there is at most one duplicated edge at any given time, and the graph obtained at the end of the process has no duplicated edges. Our choice to add a copy of a violated edge incoming to u (if such an edge exists) guarantees that the number of recursive steps is at most Δ.

Due to space limitations, more details and correctness of the deletion process are described in the full version. Overall, we prove the following theorem.

Theorem 6. *There exists a deterministic algorithm for maintaining an orientation of a fully dynamic graph on n vertices while supporting the following:*
- *The maximum out-degree is $\Delta \leq \inf_{\beta>1}\{\beta \cdot \alpha(G) + \log_\beta n\}$,*
- *The worst-case time to execute an edge insertion is $O(\Delta^2)$,*
- *The worst-case time to execute an edge deletion is $O(\Delta)$, and*
- *The worst-case number of orientations performed per update is $\Delta + 1$.*

4 A More Efficient Algorithm

In this section we present a more efficient, though more involved, algorithm that improves the insertion update time from $O(\Delta^2)$ to $O(\gamma \cdot \Delta)$, without increasing any of the other measures, at the cost of setting $\hat{\alpha}$ and $\hat{\beta}$ in advance.

An Intermediate Invariant: So far we have introduced two invariants. On one extreme, the stronger Invariant 4 guarantees that all edges are valid, and this led to our simple algorithm in Section 3. On the other extreme, the weaker Invariant 3 only guarantees that γ outgoing edges of each vertex are valid. On an intuitive level, the benefit of having the weaker Invariant 3 being maintained comes into play during the insertion process of edge (u, v) that is oriented as $u \to v$, where instead of scanning all of the outgoing edges of u looking for a violated edge, it is enough to scan only γ edges. If such a guarantee could be made to work, the insertion update time would be reduced to $O(\gamma \cdot \Delta)$. However, it is unclear how to efficiently maintain Invariant 3 as deletions take place. Specifically, when one of the γ outgoing valid edges of a vertex is deleted, it is possible that there is no other valid outgoing edge to replace it.

Our strategy is not to maintain Invariant 3 directly, but rather to define and maintain an intermediate invariant (see Invariant 7), which is stronger than Invariant 3 but still weak enough so that we only need to scan γ outgoing edges of u during the insertion process. The additional strength of the intermediate invariant will assist us in efficiently supporting deletions. Before stating the invariant, we define the following. For any $i \geq 1$, an edge (u, v) oriented as $u \to v$ is called *i-valid* if $d_{out}(v) \geq d_{out}(u) - i$; if it is not i-valid then it is *i-violated.* We also say that a vertex w is *spectrum-valid* if the set E_w of its outgoing edges can be partitioned into $q = q_w = \lceil \frac{|E_w|}{\gamma} \rceil$ sets $E_w^1, \cdots, E_w^q$ such that for each $1 \leq i \leq q$, the following holds: (1) $|E_w^i| = \gamma$ (except for the residue set E_w^q which contains the remaining $|E_w| - (q-1) \cdot \gamma$ edges, i.e., $|E_w^q| = |E_w| - (q-1) \cdot \gamma$), and (2) all edges in E_w^i are i-valid. If a vertex is not spectrum-valid then it is *spectrum-violated.*

Invariant 7. *Each vertex w is spectrum-valid.*

We will call E_w^1 (E_w^q) the first (last) set of edges for w. To give some intuition as to why Invariant 7 helps us support deletions efficiently, notice that once an edge (u, v) that is oriented as $u \to v$ is deleted and needs to be replaced, it will either be replaced by a flip of some violated incoming edge (which will become valid after the flip), or it can be replaced by one of the edges from E_u^2, as these edges were previously 2-valid, and after the deletion they are all 1-valid. We emphasize already here that during the insertion process we do not scan the γ edges of the first set (i.e., those that are guaranteed to be 1-valid prior to the insertion), but rather scan the γ (in fact, $\gamma - 1$) edges of the last set (and possibly of the set before last) that are only guaranteed to be q-valid.

In order to facilitate the use of Invariant 7, each vertex w will maintain its outgoing edges in a doubly linked list $\mathcal{L}_w$. We say that $\mathcal{L}_w$ is *valid* if for every $1 \leq i \leq q$, the edges between location $\gamma \cdot (i-1) + 1$ and location $(\gamma \cdot i)$ in the list are all i-valid. These locations for a given i are called the i-block of $\mathcal{L}_w$. So, in a valid $\mathcal{L}_w$ the first location must be 1-valid and belongs to the 1-block, the last location must be q-valid and belongs to the q-block, etc. Note that for $i = q$ the number of locations (i.e., $|E_w| - (q-1) \cdot \gamma$) may be smaller than γ. If $\mathcal{L}_w$ is not valid then it is *violated.*

We now provide an overview of the more efficient algorithms for insertion and deletion. Due to space limitations, the full details are given in the full version.

Insertions: Suppose that edge (u, v) is added to G_{i-1} thereby obtaining G_i. The process of inserting the new edge is performed as in Section 3 with the following modifications. Instead of scanning all outgoing edges of u in order to find a violated edge, we only scan the *last* $\gamma - 1$ edges in $\mathcal{L}_u$; if there are less than $\gamma - 1$ edges then we scan them all. If one of these edges, say (u, v'), is violated then we remove (u, v') from the graph, replace (u, v') with (u, v) in $\mathcal{L}_u$, and recursively insert (u, v') with the flipped orientation (just like in Section 3). If all of these edges are valid, we move them together with the new edge (u, v) to front of $\mathcal{L}_u$.

Deletions: Suppose that edge (u, v) is deleted from G_{i-1} thereby obtaining G_i. The process of deleting the edge is performed as in Section 3 with the following modifications. If an edge incoming into u, say (u, v'), is violated and is flipped (just like in Section 3), then we replace (u, v) with (u, v') in $\mathcal{L}_u$ and continue recursively to delete the original copy of (u, v'). If all incoming edges of u are valid, we remove (u, v) from $\mathcal{L}_u$.

Theorem 7. *There exists a deterministic algorithm for maintaining an orientation of a fully dynamic graph on n vertices that has arboricity at most $\hat{\alpha}$ (at all times), while supporting the following:*

- *The maximum out-degree is $\Delta \leq \hat{\beta} \cdot \hat{\alpha} + \log_{\hat{\beta}} n$,*
- *The worst-case time to execute an edge insertion is $O(\hat{\beta} \cdot \hat{\alpha} \cdot \Delta)$,*
- *The worst-case time to execute an edge deletion is $O(\Delta)$, and*
- *The worst-case number of orientations performed per update is $\Delta + 1$.*

Acknowledgments. The fourth-named author is grateful to Ofer Neiman for helpful discussions.

References

1. Alon, N., Yuster, R., Zwick, U.: Color-coding. J. ACM 42, 844–856 (1995)
2. Chrobak, M., Eppstein, D.: Planar orientations with low out-degree and compaction of adjacency matrices. Theor. Comput. Sci. 86, 243–266 (1991)
3. Brodal, G.S., Fagerberg, R.: Dynamic representation of sparse graphs. In: 6th International Workshop on Algorithms and Data Structures, WADS, pp. 342–351 (1999)
4. Kowalik, L.: Adjacency queries in dynamic sparse graphs. Inf. Process. Lett. 102, 191–195 (2007)
5. Kowalik, L., Kurowski, M.: Oracles for bounded-length shortest paths in planar graphs. ACM Transactions on Algorithms 2, 335–363 (2006)
6. Cain, J.A., Sanders, P., Wormald, N.: The random graph threshold for k-orientiability and a fast algorithm for optimal multiple-choice allocation. In: 18th Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 469–476. SIAM (2007)
7. Neiman, O., Solomon, S.: Simple deterministic algorithms for fully dynamic maximal matching. In: Proc. of 45th STOC, pp. 745–754 (2013)
8. Dvořák, Z., Tůma, V.: A dynamic data structure for counting subgraphs in sparse graphs. In: Dehne, F., Solis-Oba, R., Sack, J.-R. (eds.) WADS 2013. LNCS, vol. 8037, pp. 304–315. Springer, Heidelberg (2013)
9. Eisenstat, D., Klein, P.N., Mathieu, C.: An efficient polynomial-time approximation scheme for steiner forest in planar graphs. In: Proc. of SODA, pp. 626–638 (2012)
10. Eppstein, D.: All maximal independent sets and dynamic dominance for sparse graphs. ACM Transactions on Algorithms 5 (2009)
11. Nash-Williams, C.S.J.A.: Edge-disjoint spanning trees in finite graphs. Journal of the London Mathematical Society 36(1), 445–450 (1961)
12. Nash-Williams, C.S.J.A.: Decomposition of finite graphs into forests. Journal of the London Mathematical Society 39(1), 12 (1964)
13. Gabow, H.N., Westermann, H.H.: Forests, frames, and games: Algorithms for matroid sums and applications. Algorithmica 7, 465–497 (1992)
14. Arikati, S.R., Maheshwari, A., Zaroliagis, C.D.: Efficient computation of implicit representations of sparse graphs. Discrete Appl. Math. 78, 1–16 (1997)
15. Erickson, J.: (2006), `http://www.cs.uiuc.edu/ jeffe/teaching/datastructures/2006/problems/Bill-arboricity.pdf` (retrieved November 2013)
16. Gupta, A., Kumar, A., Stein, C.: Maintaining assignments online: Matching, scheduling, and flows. In: Chekuri, C. (ed.) SODA, pp. 468–479. SIAM (2014)