Dynamic Representations of Sparse Graphs[⋆]

Gerth Stølting Brodal and Rolf Fagerberg

BRICS[⋆⋆], Department of Computer Science, University of Aarhus,
DK-8000 Århus C, Denmark
{gerth,rolf}@brics.dk

Abstract. We present a linear space data structure for maintaining graphs with bounded arboricity—a large class of sparse graphs containing e.g. planar graphs and graphs of bounded treewidth—under edge insertions, edge deletions, and adjacency queries.
The data structure supports adjacency queries in worst case $\mathcal{O}(c)$ time, and edge insertions and edge deletions in amortized $\mathcal{O}(1)$ and $\mathcal{O}(c+\log n)$ time, respectively, where n is the number of nodes in the graph, and c is the bound on the arboricity.

1 Introduction

A fundamental operation on graphs is, given two nodes u and v, to tell whether or not the edge (u, v) is present in the graph. We denote such queries *adjacency queries.*

For a graph $G = (V, E)$, let $n = |V|$ and $m = |E|$. Two standard ways to represent graphs are adjacency matrices and adjacency lists [4, 16]. In the former case, adjacency queries can be answered in $\mathcal{O}(1)$ time, but the space required is $\Theta(n^2)$ bits, which is super-linear for sparse graphs. In the latter case, the space is reduced to $\mathcal{O}(m)$ words, but now adjacency queries involve searching neighbor lists, and these may be of length $\Theta(n)$. Using sorted adjacency lists, this gives $\mathcal{O}(\log n)$ worst case time for adjacency queries.

A third approach is to use perfect hashing, which gives $\mathcal{O}(1)$ query time and $\mathcal{O}(m)$ space, but has the drawback that the fastest known methods for constructing linear space hash tables either use randomization [5] or spend $\Theta(m^{1+\epsilon})$ time [9].

There have been a number of papers on representing a sparse graph succinctly while allowing adjacency queries in $\mathcal{O}(1)$ time, among these [1, 3, 8, 10, 14, 15]. Various authors emphasize different aspects, including minimizing the exact number of bits used, construction in linear time and versions for parallel and distributed environments. However, all data structures proposed have been for static graphs only.

⋆ Partially supported by the ESPRIT Long Term Research Program of the EU under contract 20244 (project ALCOM-IT).

⋆⋆ BRICS (Basic Research in Computer Science), a Centre of the Danish National Research Foundation.

In this paper, we study the *dynamic* version of the problem, where edges can be inserted and deleted in the graph. This case is stated as an open problem in [8].

Like [1], we consider the class of graphs having *bounded arboricity*. The *arboricity* c of a graph $G = (V, E)$ is defined by

$$c = \max_J \frac{|E(J)|}{|V(J)| - 1} ,$$

where J is any subgraph of G with $|V(J)| \geq 2$ nodes and $|E(J)|$ edges. This class contains graphs with bounded genus g (since $m \leq 6(g-1) + 3n$ by Euler's formula), in particular planar graphs (where $c \leq 3$, since $g = 0$), as well as graphs of bounded degree d ($c \leq \lfloor d/2 \rfloor + 1$), and graphs of bounded treewidth t ($c \leq t$). Intuitively, the graphs of bounded arboricity is the class of uniformly sparse graphs.

More precisely, we consider the problem of maintaining an undirected graph $G = (V, E)$ of arboricity at most c under the operations

- $\mathsf{Adjacent}(u, v)$, return true if and only if $(u, v) \in E$,
- $\mathsf{Insert}(u, v)$, $E := E \cup \{(u, v)\}$,
- $\mathsf{Delete}(u, v)$, $E := E \setminus \{(u, v)\}$,
- $\mathsf{Build}(V, E)$, $G := (V, E)$.

In this paper, we present an $\mathcal{O}(m+n)$ space data structure for storing graphs of arboricity bounded by c. The data structure supports $\mathsf{Adjacent}(u, v)$ in worst case $\mathcal{O}(c)$ time, $\mathsf{Insert}(u, v)$ in amortized $\mathcal{O}(1)$ time, $\mathsf{Delete}(u, v)$ in amortized $\mathcal{O}(c + \log n)$ time, and $\mathsf{Build}(V, E)$ in amortized $\mathcal{O}(m + n)$ time.

The data structure is a slight variation of the adjacency list representation of graphs, with a simple—almost canonical—maintenance algorithm. Our proof of complexity is by a reduction, which shows that the analysis of *any* algorithm for the problem of maintaining such adjacency lists carries over to the presented algorithm, with the right choice of parameters (see Lemma 1 for details).

It is assumed that c is known by the algorithm. It is also assumed that at no point is an edge inserted, which already is present or which violates the arboricity constraint—i.e. it is the responsibility of the application using the data structure to guarantee that the bounded arboricity constraint is satisfied. For a given graph, a 2-approximation of the arboricity can easily be computed in $\mathcal{O}(m + n)$ time [1]. More complicated algorithms calculating the exact value are presented in [6]. The extension of the algorithm presented to the case of unknown c is discussed in Sect. 4.

The graphs considered are simple, undirected graphs. However, the data structure can easily be extended to allow the annotation of edges and nodes with auxiliary information, allowing graphs with self-loops, multiple edges, or directed edges to be represented. For simplicity, we assume all nodes to be present from the beginning. It is straightforward to extend the structure to allow insertions of new nodes.

When undirected graphs are represented by adjacency lists, an edge normally appears in the list of both of its endpoints. A basic observation, used in [1, 8], is that if each edge is stored in the list of only *one* of its two endpoints, then adjacency queries $\mathsf{Adjacent}(u, v)$ can still be answered by searching the adjacency list of both u and v. The advantage gained is the possibility of nodes having short adjacency lists, even if they have high degree.

Storing each edge at only one of its endpoints is equivalent to assigning an orientation to all edges of the graph—view an edge (u, v) as going from u to v if v is stored in the adjacency list of u. We will therefore refer to such a distribution of edges into adjacency lists as an orientation of the graph. Formally, an *orientation* of an undirected graph $G = (V, E)$, is a directed graph $\bar{G} = (V, \bar{E})$ on the same set of nodes, where $\bar{E}$ is equal to E when the elements of $\bar{E}$ are seen as unordered pairs.

If adjacency queries are to take constant time, the adjacency lists should be of bounded length. In other words, we are interested in orientations where the outdegree of each node is bounded by some constant Δ. We call such orientations for *Δ-orientations.*

For graphs of arboricity c, a c-orientation always exists. This follows from the classical characterization of such graphs (which also gives rise to their name) by Nash-Williams:

Theorem 1 (Nash-Williams [2, 11, 12]). *A graph $G = (V, E)$ has arboricity c if and only if c is the smallest number of sets $E_1, \ldots, E_c$ that E can be partitioned into, such that each subgraph (V, E_i) is a forest.*

If we arbitrarily choose a root in each tree in each of the c forests, and orient all edges in the trees towards the roots, then each node has outdegree at most one in each forest and hence outdegree at most c in the entire graph.

Finding such a decomposition into exactly c forests is non-trivial. An algorithm was given in [13] which takes $\mathcal{O}(n^2 m \log^2 n)$ time for graphs of arboricity c. This was later improved to $\mathcal{O}(cn\sqrt{m + cn \log n})$ in [6]. For planar graphs, where $c = 3$, an $\mathcal{O}(n \log n)$ algorithm appears in [7]. However, for adjacency queries to take $\mathcal{O}(c)$ time, it is sufficient to find an $\mathcal{O}(c)$-orientation. A simple $\mathcal{O}(m + n)$ time algorithm computing a $(2c - 1)$-orientation was described in [1].

In this paper, we address the question of how to maintain $\mathcal{O}(c)$-orientations during insertions and deletions of edges on graphs whose arboricity stays bounded by some constant c.

Note that, in a sense, the class of graphs of bounded arboricity is the maximal class of graphs for which Δ-orientations exist, as any graph for which a Δ-orientation exists has arboricity at most 2Δ, as $|E(J)| \leq \Delta |V(J)| \leq 2\Delta(|V(J)| - 1)$ for all subgraphs J with $|V(J)| \geq 2$.

The rest of this paper is organized as follows: In Sect. 2, we present our algorithm for maintaining Δ-orderings. In Sect. 3, we first prove that the algorithm inherits the amortized analysis of any algorithm for maintaining δ-orderings, provided $\Delta \geq 2\delta$. We then give a non-constructive proof of existence of an algorithm with the desired amortized complexity. Section 4 contains further comments on the problem of maintaining Δ-orderings, as well as on the presented algorithm. Finally, Sect. 5 lists some open problems.

2 The algorithm

As described in the introduction, we reduce the problem of achieving constant time adjacency queries for graphs with arboricity at most c to the problem of assigning orientations to the edges such that all nodes have outdegree $\mathcal{O}(c)$. Our data structure is simply an adjacency list representation of the directed graph.

Our maintenance algorithm guarantees that all nodes have outdegree at most Δ, where Δ is a parameter depending on the arboricity c. In Sect. 3 we show that $\Delta = 4c$ results in the time bounds stated in the introduction.

Pseudo code for our maintenance algorithm is given in Fig. 1. The list of nodes reachable by a directed edge from u is denoted $\mathsf{adj}[u]$. Nodes with degree larger than Δ are stored on a stack S, and a node v is pushed onto S when its outdegree increases from Δ to $\Delta + 1$.

A query $\mathsf{Adjacent}(u, v)$ is answered by searching the adjacency lists of u and v. In $\mathsf{Insert}(u, v)$, v is first inserted into the adjacency list of u. If u gets outdegree $\Delta+1$, repeatedly a node w with outdegree larger than Δ is picked, and the orientation of all outgoing edges from w is changed, such that w gets outdegree zero. This continues until all nodes have outdegree at most Δ. The operation $\mathsf{Delete}(u, v)$ simply removes the corresponding directed edge, and $\mathsf{Build}(V, E)$ incrementally inserts the edges $(u, v) \in E$ in any order, using $\mathsf{Insert}(u, v)$.

```
proc Adjacent($u, v$)
  return ($v \in$ adj[$u$] or $u \in$ adj[$v$])

proc Delete($u, v$)
  adj[$u$] := adj[$u$] $\setminus \{v\}$
  adj[$v$] := adj[$v$] $\setminus \{u\}$

proc Build($V, E$)
  forall $v \in V$
    adj[$v$] := $\emptyset$
  forall $(u, v) \in E$
    Insert($u, v$)

proc Insert($u, v$)
  adj[$u$] := adj[$u$] $\cup \{v\}$
  if |adj[$u$]| = $\Delta + 1$
    $S := \{u\}$
    while $S \neq \emptyset$
      $w$ := Pop($S$)
      foreach $x \in$ adj[$w$]
        adj[$x$] := adj[$x$] $\cup \{w\}$
        if |adj[$x$]| = $\Delta + 1$
          Push($S, x$)
      adj[$w$] := $\emptyset$
```

Fig. 1. Pseudo code for the procedures.

3 Analysis

We first give some definitions. An *arboricity c preserving* sequence of edge insertions and edge deletions on a graph G, initially of arboricity at most c, is a sequence of operations where the arboricity stays bounded by c during the entire sequence. Given two orientations $(V, \bar{E}_i)$ and $(V, \bar{E}_{i+1})$, the number of *edge reorientations* between $(V, \bar{E}_i)$ and $(V, \bar{E}_{i+1})$ is the number of edges which are present in both graphs but with different orientations, i.e. edges (u, v) where $(u, v) \in \bar{E}_i$ and $(v, u) \in \bar{E}_{i+1}$ or vice versa.

The following lemma allows us to compare the presented algorithm with any algorithm based on assigning orientations to the edges.

Lemma 1 (Main reduction). *Given an arboricity c preserving sequence σ of edge insertions and deletions on an initially empty graph, let G_i be the graph after the i'th operation, and let k be the number of edge insertions.*

If there exists a sequence $\bar{G}_0, \bar{G}_1, \ldots, \bar{G}_{|\sigma|}$ of δ-orientations with at most r edge reorientations in total, then the algorithm performs at most

$$(k + r)\frac{\Delta + 1}{\Delta + 1 - 2\delta}$$

edge reorientations in total on the sequence σ, provided $\Delta \geq 2\delta$.

Proof. We analyze the algorithm by comparing the edge orientations assigned by it to the δ-orientations $\bar{G}_i = (V, \bar{E}_i)$. An edge $(u, v) \in E_i$ is denoted *good* if (u, v) by the algorithm has been assigned the same orientation as in $\bar{E}_i$, i.e. $v \in \text{adj}[u]$ and $(u, v) \in \bar{E}_i$ or $u \in \text{adj}[v]$ and $(v, u) \in \bar{E}_i$. Otherwise $(u, v) \in E_i$ is *bad.* To analyze the number of reorientations done by the algorithm, we consider the following non-negative potential:

$$\Psi = \text{the number of bad edges in the current } E_i \ .$$

Initially, $\Psi = 0$. Each of the k edges inserted and r edge reorientations in the δ-orientations $\bar{G}_i$ increases Ψ by at most one. Deleting edges cannot increase Ψ. Consider an iteration of the **while** loop where the orientation is changed of the outgoing edges of a node w with outdegree at least $\Delta + 1$. At most δ outgoing edges of w can be good (since the orientations $\bar{G}_i$ are δ-orientations). By changing the orientation of the outgoing edges of w, at most δ good edges become bad, and the remaining at least $\Delta + 1 - \delta$ bad edges become good. It follows that Ψ decreases by at least $\Delta + 1 - 2\delta$ in each iteration of the **while** loop. The number of iterations of the **while** loop is therefore at most $(k + r)/(\Delta + 1 - 2\delta)$. The total number of times a good edge is made bad in the **while** loop is at most $\delta(k + r)/(\Delta + 1 - 2\delta)$, implying that at most $k + r + \delta(k + r)/(\Delta + 1 - 2\delta)$ times a bad edge is made good in the **while** loop. In total, the algorithm does at most $(k+r)(1+2\delta/(\Delta+1-2\delta))$ edge reorientations. Rearranging gives the result. □

Lemma 2. *Let $G = (V, E)$ be a graph with arboricity at most c, let $\bar{G} = (V, \bar{E})$ an orientation of G, and let $\delta > c$. In $\bar{G}$, if $u \in V$ has outdegree at least δ then there exists a node v with outdegree less than δ and a directed path from u to v containing at most $\lceil \log_{\delta/c} |V| \rceil$ edges.*

Proof. In the following, we consider the graph $\bar{G}$, and let $V_i \subseteq V$ be the set of nodes reachable from u by directed paths containing at most i edges, i.e. $V_0 = \{u\}$ and $V_{i+1} = V_i \cup \{w \in V \mid \exists w' \in V_i : (w', w) \in \bar{E}\}$.

For $i \geq 1$, we prove by induction that if all nodes in V_i have outdegree at least δ, then $|V_i| > (\delta/c)^i$. Since V_1 contains u and at least δ nodes adjacent to u, we have $|V_1| \geq 1 + \delta > \delta/c$. For $i \geq 1$, assume $|V_i| > (\delta/c)^i$ and all nodes in V_i have outdegree at least δ, i.e. the total number of outgoing edges from nodes in V_i is at least $\delta|V_i|$, and by definition of V_{i+1} these edges connect nodes in V_{i+1}. Because any subgraph (V', E') of G also has arboricity at most c, i.e. $|V'| \geq 1 + |E'|/c$, we have $|V_{i+1}| \geq 1 + (\delta|V_i|)/c > (\delta/c)^{i+1}$.

If all nodes in V_i have outdegree at least δ then $|V| \geq |V_i| > (\delta/c)^i$, from which we have $i < \log_{\delta/c} |V|$ and the lemma follows. □

Lemma 3. *Given an arboricity c preserving sequence of edge insertions and deletions on an initially empty graph, there for any $\delta > c$ exists a sequence of δ-orientations, such that*

1. *for each edge insertion there are no edge reorientations,*
2. *for each edge deletion there are at most $\lceil \log_{\delta/c} |V| \rceil$ edge reorientations.*

Proof. Let k denote the number of edge insertions and deletions, and let $G_i = (V, E_i)$ denote the graph after the i'th operation, for $i = 0, \ldots, k$, with $E_0 = \emptyset$. Since G_k has arboricity at most c, we by Theorem 1 have a c-orientation $\bar{G}_k$ of G_k, which is a δ-orientation since $\delta \geq c$.

We now construct δ-orientations $\bar{G}_i = (V, \bar{E}_i)$ inductively in decreasing order on i. If G_{i+1} follows from G_i by inserting edge (u, v), i.e. G_i follows by deleting edge (u, v) from G_{i+1}, we let $\bar{E}_i = \bar{E}_{i+1} \setminus \{(u, v), (v, u)\}$. If $\bar{G}_{i+1}$ is a δ-orientation, then $\bar{G}_i$ is also a δ-orientation. If G_{i+1} follows from G_i by deleting edge (u, v), i.e. G_i follows from G_{i+1} by inserting edge (u, v), there are two cases to consider. If u in $\bar{G}_{i+1}$ has outdegree less than δ, then we set $\bar{E}_i = \bar{E}_{i+1} \cup \{(u, v)\}$. Otherwise u has outdegree δ in $\bar{G}_{i+1}$, and by Lemma 2 there exists a node v' with outdegree less than δ in $\bar{G}_{i+1}$, and a directed path in $\bar{G}_{i+1}$ from u to v' containing at most $\lceil \log_{\delta/c} |V| \rceil$ edges. By letting $\bar{E}_i$ be $\bar{E}_{i+1}$ with the orientation of the edges in p reversed, plus the edge (u, v), only the outdegree of v' increases by one. In both cases $\bar{G}_i$ is a δ-orientation if $\bar{G}_{i+1}$ is a δ-orientation. □

Theorem 2. *In an arboricity c preserving sequence of operations starting with an empty graph, the algorithm for $\Delta/2 \geq \delta > c$ supports* $\mathsf{Insert}(u, v)$ *in amortized $\mathcal{O}(\frac{\Delta+1}{\Delta+1-2\delta})$ time,* $\mathsf{Build}(V, E)$ *in amortized $\mathcal{O}(|V| + |E|\frac{\Delta+1}{\Delta+1-2\delta})$ time,* $\mathsf{Delete}(u, v)$ *in amortized $\mathcal{O}(\Delta + \frac{\Delta+1}{\Delta+1-2\delta} \log_{\delta/c} |V|)$ time, and* $\mathsf{Adjacent}(u, v)$ *in worst case $\mathcal{O}(\Delta)$ time.*

Proof. The worst-case time for $\mathsf{Delete}(u, v)$ and $\mathsf{Adjacent}(u, v)$ are clearly $\mathcal{O}(\Delta)$, and the worst case time for $\mathsf{Insert}(u, v)$ is $\mathcal{O}(1)$ plus the time spent in the **while** loop for reorientating edges, and since $\mathsf{Build}(V, E)$ is implemented using $\mathsf{Insert}(u, v)$, this takes $O(|V| + |E|)$ time plus the time spent in the **while** loop for reorientating edges.

Combining Lemmas 1 and 3 gives that for any δ satisfying $\Delta/2 \geq \delta > c$, a sequence of a edge insertions and b edge deletions requires at most $(a + b\lceil \log_{\delta/c} |V| \rceil)\frac{\Delta+1}{\Delta+1-2\delta}$ edge reorientations. This implies that the amortized number of edge reorientations for an edge insertion is at most $\frac{\Delta+1}{\Delta+1-2\delta}$, and that the amortized number of edge reorientations for an edge deletion is at most $\frac{\Delta+1}{\Delta+1-2\delta}\lceil\log_{\delta/c}|V|\rceil$. □

One possible choice of parameters in Theorem 2 is $\Delta = 4c$ and $\delta = \frac{3}{2}c$, which gives:

Theorem 3. *In an arboricity c preserving sequence of operations starting with an empty graph, the algorithm with $\Delta = 4c$ supports* $\mathsf{Insert}(u, v)$ *in amortized $\mathcal{O}(1)$ time,* $\mathsf{Build}(V, E)$ *in amortized $\mathcal{O}(|V| + |E|)$ time,* $\mathsf{Delete}(u, v)$ *in amortized $\mathcal{O}(c + \log|V|)$ time, and* $\mathsf{Adjacent}(u, v)$ *in worst case $\mathcal{O}(c)$ time.*

4 Discussion

By Theorem 1 there exists a c-orientation of any graph of arboricity bounded by a constant c. Interestingly, by Theorem 4 below such an orientation cannot be maintained in less than linear time per operation in the dynamic case.

Theorem 4. *Let $\mathcal{A}$ be an algorithm maintaining orientations on edges during insertion and deletion of edges in a graph with n nodes. If $\mathcal{A}$ guarantees that no node ever has outdegree larger than c, provided that the arboricity of the graph stays bounded by c, then for at least one of the operations insert and delete, $\mathcal{A}$ can be forced to change the orientation of $\Omega(n/c^2)$ edges, even when considering amortized complexity.*

Proof. For any even integer $k > 2$, consider the graph on $c \cdot k$ nodes shown below (with $c = 4$). Let the j'th node in the i'th row be labeled $v_{i,j}$, where $i = 0, 1, \ldots, c-1$ and $j = 0, 1, \ldots, k-1$. The graph can be decomposed into c^2 edge-disjoint paths $T_{a,b}$, for $a, b = 0, 1, \ldots, c-1$, where $T_{a,b}$ is the path through the points $v_{a,0}$, $v_{b,1}, v_{a,2}$, $v_{b,3}$, ... , $v_{b,k-1}$. In Figure 2, the path $T_{0,2}$ is highlighted.

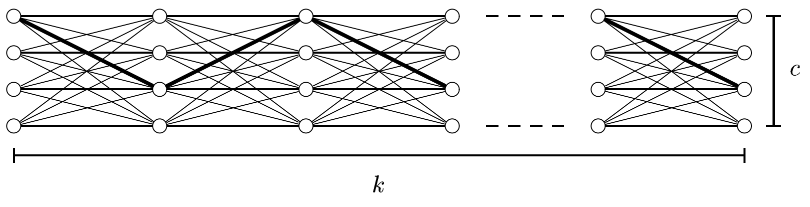

Fig. 2.

To this graph, we c times add $c - 1$ edges between nodes at the rightmost and leftmost ends of the graph in such a way that c of the paths $T_{a,b}$ are concatenated into one path. More precisely, for each $r = 0, 1, \ldots, c-1$, the c paths

$T_{i,i+r \bmod c}$, $i = 0, 1, \ldots, c-1$, are concatenated into one path U_r by adding the edges $(v_{i+r \bmod c,k-1}\ ,\ v_{i+1,0})$, $i = 0, 1, \ldots, c-2$. To exemplify, U_0 consists of the c horizontal paths $T_{i,i}$, $i = 0, 1, \ldots, c-1$, concatenated by the $c-1$ new edges $(v_{0,k-1}\ ,\ v_{1,0}), (v_{1,k-1}\ ,\ v_{2,0}), \ldots, (v_{c-2,k-1}\ ,\ v_{c-1,0})$.

As the resulting graph G is composed of the c edge-disjoint paths U_r, $r = 0, 1, \ldots, c-1$, it has arboricity at most c. Its number of nodes n is $c \cdot k$ and its number of edges m is $c^2(k-1) + c(c-1) = c^2k - c$. Counting edges gives that for any orientation on the edges of this graph with an outdegree of c or less for all nodes, at most c of the nodes can have outdegree strictly less than c. Hence there is a contiguous section of $\Omega(k/c)$ columns of the graph where all nodes have outdegree equal to c.

Let t be the index of a column in the middle of this section. Now remove the edge $(v_{0,k-1}\ ,\ v_{1,0})$ and add the edge $(v_{0,t}\ ,\ v_{1,t})$. The resulting graph still has arboricity at most c, as it can be decomposed into the paths $U_1, U_2, \ldots, U_{c-1}$ and a tree $\tilde{U}_0$ (derived from U_0 by the described change of one edge), all of which are edge-disjoint.

One of the nodes $v_{0,t}$ and $v_{1,t}$ now has outdegree $c+1$, and the algorithm must change the orientation of some edges. Note that the change of orientation of one edge effectively moves a count of one between the outdegrees of two neighboring nodes. As all paths in the graph between the overflowing node and a node with outdegree strictly less than c have length $\Omega(k/c) = \Omega(n/c^2)$, at least this number of edges must have their orientation changed. As we can return to the graph G by deleting $(v_{0,t}\ ,\ v_{1,t})$ and inserting $(v_{0,k-1}\ ,\ v_{1,0})$ again, the entire process can be repeated, and hence the lower bound also holds when considering amortized complexity. $\square$

In our algorithm, $\Delta \geq 2c + 2$ is necessary for the analysis of Lemma 2. A theoretically interesting direction for further research is to determine exactly how the complexity of maintaining a Δ-orientation changes when Δ ranges from c to $2c$. Note that Lemma 2 in a non-constructive way shows that $(c+1)$-orientations can be maintained in a logarithmic number of edge reorientations per operation.

The dependency on c in the time bounds of Theorems 2 and 3 can be varied somewhat by changing the implementation of the adjacency lists. The bounds stated hold for unordered lists. If balanced search trees are used, the occurrences of c in the time bounds in Theorem 3 for $\mathsf{Adjacent}(u, v)$ and $\mathsf{Delete}(u, v)$ become $\log c$, at the expense of the amortized complexity of $\mathsf{Insert}(u, v)$ increasing from $\mathcal{O}(1)$ to $\mathcal{O}(\log c)$. If we assume that we have a pointer to the edge in question when performing $\mathsf{Delete}(u, v)$ (i.e. a pointer to the occurrence of v in u's adjacency list, if the edge has been directed from u to v), then the dependency on c can be removed from the time bound for this operation.

In the algorithm, we have so far assumed that the bound c on the arboricity is known. If this is not the case, an adaptive version can be achieved by letting the algorithm continually count the number of edge reorientations that it makes. If the count at some point exceeds the bound in Theorem 2 for the current value of Δ, the algorithm doubles Δ, resets the counter and performs a $\mathsf{Build}(V, E)$

operation. As $\mathsf{Build}(V, E)$ takes linear time, this scheme only adds an additive term of $\log c$ to the amortized complexity of $\mathsf{Insert}(u, v)$. If the algorithm must adapt to decreasing as well as increasing arboricity, the sequence of operations can be divided into phases of length $\Theta(|E|)$, after which the value of Δ is reset to some small value, the counter is reset, and a $\mathsf{Build}(V, E)$ is done.

5 Open problems

An obvious open question is whether both $\mathsf{Insert}(u, v)$ and $\mathsf{Delete}(u, v)$ can be supported in amortized $\mathcal{O}(1)$ time. Note that any improvement of Lemma 3, e.g. by an algorithm maintaining edge orientations in an amortized sublogarithmic number of edge reorientations per operation (even if it takes, say, exponential *time*), will imply a correspondingly improved analysis of our algorithm, by Lemma 1. Conversely, by the negation of this statement, lower bounds proved for our algorithm will imply lower bounds on all algorithms maintaining edge orientations.

A succinct graph representation which supports $\mathsf{Insert}(u, v)$ and $\mathsf{Delete}(u, v)$ efficiently in the worst case sense would also be interesting.

References

1. Srinivasa R. Arikati, Anil Maheshwari, and Christos D. Zaroliagis. Efficient computation of implicit representations of sparse graphs. *Discrete Applied Mathematics*, 78:1–16, 1997.
2. Boliong Chen, Makoto Matsumoto, Jian Fang Wang, Zhong Fu Zhang, and Jian Xun Zhang. A short proof of Nash-Williams' theorem for the arboricity of a graph. *Graphs Combin.*, 10(1):27–28, 1994.
3. Chuang, Garg, He, Kao, and Lu. Compact encodings of planar graphs via canonical orderings and multiple parentheses. In *ICALP: Annual International Colloquium on Automata, Languages and Programming*, 1998.
4. Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*, chapter 23. MIT Press, Cambridge, Mass., 1990.
5. Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *Journal of the Association for Computing Machinery*, 31(3):538–544, 1984.
6. Harold N. Gabow and Herbert H. Westermann. Forests, frames, and games: Algorithms for matroid sums and applications. *Algorithmica*, 7:465–497, 1992.
7. Grossi and Lodi. Simple planar graph partition into three forests. *Discrete Applied Mathematics*, 84:121–132, 1998.
8. Sampath Kannan, Moni Naor, and Steven Rudich. Implicit representation of graphs. *SIAM Journal on Discrete Mathematics*, 5(4):596–603, 1992.
9. Peter Bro Miltersen. Error correcting codes, perfect hashing circuits, and deterministic dynamic dictionaries. In *Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 556–563, 1998.
10. J. Ian Munro and Venkatesh Raman. Succinct representation of balanced parentheses, static trees and planar graphs. In *38th Annual Symposium on Foundations of Computer Science*, pages 118–126, 20–22 October 1997.

11. C. St. J. A. Nash-Williams. Edge-disjoint spanning trees of finite graphs. *The Journal of the London Mathematical Society*, 36:445–450, 1961.
12. C. St. J. A. Nash-Williams. Decomposition of finite graphs into forests. *The Journal of the London Mathematical Society*, 39:12, 1964.
13. J. C. Picard and M. Queyranne. A network flow soloution to some non-linear 0-1 programming problems, with applications to graph theory. *Networks*, 12:141–160, 1982.
14. M. Talamo and P. Vocca. Compact implicit representation of graphs. In *Graph-Theoretic Concepts in Computer Science*, volume 1517 of *Lecture Notes in Computer Science*, pages 164–176, 1998.
15. G. Turan. Succinct representations of graphs. *Discrete Applied Math*, 8:289–294, 1984.
16. Jan van Leeuwen. Graph algorithms. In *Handbook of Theoretical Computer Science, vol. A: Algorithms and Complexity*, pages 525–631. North-Holland Publ. Comp., Amsterdam, 1990.