Dynamic Dominators and Low-High Orders in DAGs

Loukas Georgiadis
Department of Computer Science & Engineering, University of Ioannina, Greece
loukas@cs.uoi.gr

Konstantinos Giannis
Department of Computer Science & Engineering, University of Ioannina, Greece
giannis_konstantinos@outlook.com

Giuseppe F. Italiano
LUISS University, Rome, Italy
gitaliano@luiss.it

Aikaterini Karanasiou
Università di Roma "Tor Vergata", Italy
aikaranasiou@gmail.com

Luigi Laura
LUISS University, Rome, Italy
llaura@luiss.it

Abstract

We consider practical algorithms for maintaining the dominator tree and a low-high order in directed acyclic graphs (DAGs) subject to dynamic operations. Let G be a directed graph with a distinguished start vertex s. The dominator tree D of G is a tree rooted at s, such that a vertex v is an ancestor of a vertex w if and only if all paths from s to w in G include v. The dominator tree is a central tool in program optimization and code generation, and has many applications in other diverse areas including constraint programming, circuit testing, biology, and in algorithms for graph connectivity problems. A low-high order of G is a preorder of D that certifies the correctness of D, and has further applications in connectivity and path-determination problems.

We first provide a practical and carefully engineered version of a recent algorithm [ICALP 2017] for maintaining the dominator tree of a DAG through a sequence of edge deletions. The algorithm runs in $O(mn)$ total time and $O(m)$ space, where n is the number of vertices and m is the number of edges before any deletion. In addition, we present a new algorithm that maintains a low-high order of a DAG under edge deletions within the same bounds. Both results extend to the case of *reducible* graphs (a class that includes DAGs). Furthermore, we present a fully dynamic algorithm for maintaining the dominator tree of a DAG under an intermixed sequence of edge insertions and deletions. Although it does not maintain the $O(mn)$ worst-case bound of the decremental algorithm, our experiments highlight that the fully dynamic algorithm performs very well in practice. Finally, we study the practical efficiency of all our algorithms by conducting an extensive experimental study on real-world and synthetic graphs.

2012 ACM Subject Classification Theory of computation → Dynamic graph algorithms

Keywords and phrases Connectivity, dominators, low-high orders

Digital Object Identifier 10.4230/LIPIcs.ESA.2019.50

Funding *Giuseppe F. Italiano*: Work partially supported by MIUR, the Italian Ministry for Education, University and Research, under PRIN Project AHeAD (Efficient Algorithms for HArnessing Networked Data).

Acknowledgements We thank the anonymous reviewers for several comments that helped us improve the presentation of our paper.

1 Introduction

Dynamic graph algorithms have been extensively studied for several decades, and many important results have been achieved for fundamental problems, including connectivity, minimum spanning tree, transitive closure, shortest paths (see, e.g., the survey in [13]). Typically, the goal of a dynamic graph algorithm is to update the solution of a problem, following the insertion or deletion of an edge, as quickly as possible (usually much faster than recomputing from scratch). A dynamic graph problem is said to be *fully dynamic* if it is required to process both insertions and deletions of edges, *incremental* if it requires to process edge insertions only and *decremental* if it requires to process edge deletions only. Here we consider two decremental problems in directed graphs, namely maintaining the dominator tree and a low-high order of a flow graph.

A *flow graph* $G = (V, E, s)$ is a directed graph (digraph) with a distinguished start vertex $s \in V$. A vertex v is *reachable* in G if there is a path from s to v; v is *unreachable* if no such path exists. The *dominator relation* in G is defined for the set of reachable vertices as follows. A vertex v is a *dominator* of a vertex w (v *dominates* w) if every path from s to w contains v; v is a *proper dominator* of w if v dominates w and $v \neq w$. The dominator relation in G can be represented by a tree rooted at s, the *dominator tree* D, such that v dominates w if and only if v is an ancestor of w in D. See Figure 1. If $w \neq s$ is reachable, we denote by $d(w)$ the parent of w in D. The dominator tree of a flow graph can be computed in linear time [2, 8, 14, 15]. The dominator tree is a central tool in program optimization and code generation [11], and it has many applications in other diverse areas including constraint programming [40], circuit testing [4], biology [1, 29], memory profiling [38], the analysis of diffusion networks [28], and in connectivity problems [17, 18, 21, 22, 24, 31, 32, 33, 34].

A *low-high order of* G [25] is a preorder of the dominator tree D such that for all reachable vertices $v \neq s$, $(d(v), v) \in E$ or there are two edges $(u, v), (w, v) \in E$, where u and w are reachable, w is not a descendant of v in D, and $u < v < w$ in low-high order. See Figure 1. Every flow graph G has a low-high order, computable in linear-time [25]. Low-high orders provide a correctness certificate for dominator trees that is straightforward to verify [46]. By augmenting an algorithm that computes the dominator tree D of a flow graph G so that it also computes a low-high order of G, one obtains a *certifying algorithm* to compute D. Low-high orders also have applications in path-determination problems [45] and in fault-tolerant network design [5, 6, 26].

In this paper we consider how to maintain the dominator tree and a low-high order of acyclic flow graphs subject to dynamic operations. We believe that acyclic graphs are not a significant restriction, since several real-world networks, such as certain types of biological networks, are acyclic [29]. Furthermore, our results extend to reducible flow graphs (defined below), a class that includes acyclic flow graphs. Reducible flow graphs are important in program optimization since one notion of a "structured" program is that its flow graph is reducible. The dynamic dominator problem arises in various applications, such as data flow analysis and compilation [10, 16]. Moreover, dynamic dominators can be used for dynamically testing various connectivity properties in digraphs, such as 2-vertex connectivity, strong bridges and strong articulation points [32].

The problem of updating the dominator relation has been studied for several decades (see, e.g., [3, 9, 10, 20, 23, 41, 42]). While for the incremental dominators problem there are simple algorithms that achieve total $O(mn)$ running time for processing a sequence of edge insertions in a flow graph with n vertices, where m is the number of edges after all insertions [3, 10, 23], the decremental version seems much harder. Cicerone et al. [10] achieved a total $O(mn)$ update bound for the decremental problem in reducible flow graphs, where

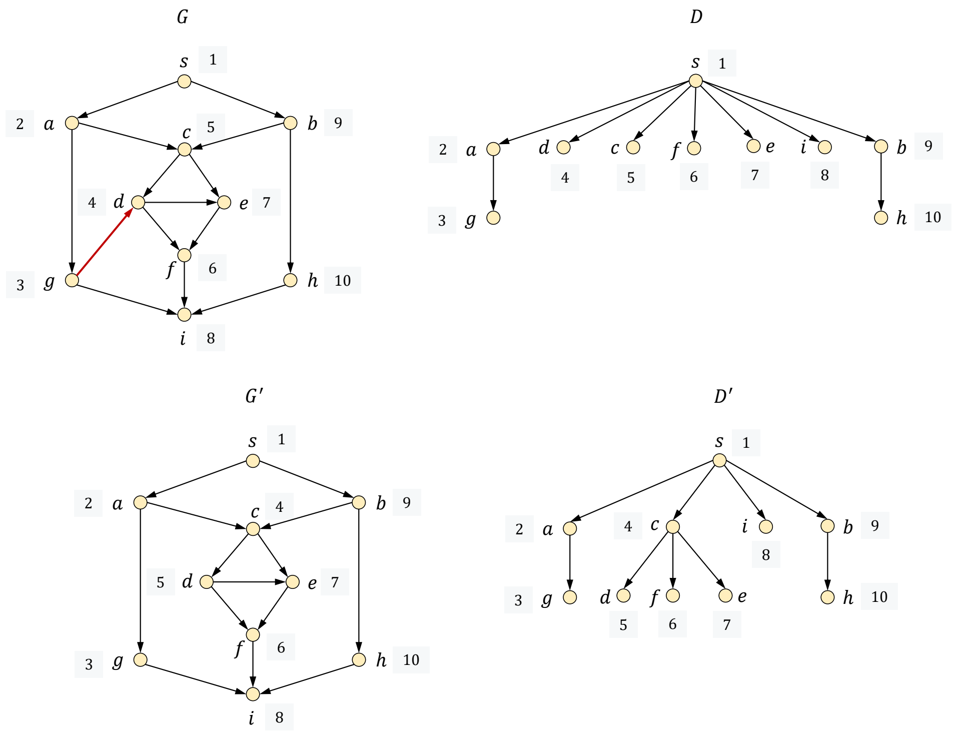

Figure 1 (Top) A flow graph G and its dominator tree D. The numbers correspond to a preorder numbering of D that is a low-high order of G. (Bottom) The flow graph G' and its dominator tree D' after the deletion edge (g, d).

m is the initial number of edges, but using $O(n^2)$ space. For general digraphs, Georgiadis et al. [20] presented an algorithm that can process a sequence of edge deletions in a flow graph in $O(mn \log n)$ total time and $O(n^2 \log n)$ space, and can answer dominance queries in constant time. For reducible flow graphs, Georgiadis et al. [20] presented an algorithm that achieves $O(mn)$ total running time using only $O(m + n)$ space. A conditional lower bound in [20] suggests that it might be hard to substantially improve the $O(mn)$ update bounds in the partial dynamic (incremental or decremental) problem of maintaining the dominator tree, even for acyclic flow graphs. As the algorithms in [20] are quite sophisticated, their implementation was a challenging task. Nevertheless, we show here that this is really worth the effort, since their efficient implementation performs very well in practice. To produce an implementation of practical value, we performed a careful engineering and choice of data structures, including a data structure for an extension of the dynamic list order maintenance problem [7, 12] and a data structure for maintaining and updating derived edges [25]. To assess the merits of our implementation in practical scenarios, we conducted a thorough experimental study.

As a second contribution, we show that we can maintain decrementally a low-high order of a reducible flow graph in $O(mn)$ total time. This implies the first decremental *certifying algorithm* [39] for computing dominators in $O(mn)$ total time in reducible flow graphs. It also immediately provides $O(mn)$-time algorithms for the following problems:

- A data structure that maintains an acyclic flow graph G decrementally, and answers the following queries in constant time: (i) For any two query vertices v and w, find a path π_{sv} from s to v and a path π_{sw} from s to w that are maximally vertex-disjoint, i.e., such that π_{sv} and π_{sw} share only the common dominators of v and w. We can output these paths in $O(|\pi_{sv}| + |\pi_{sw}|)$ time. (ii) For any two query vertices v and w, find a path π_{sv} from s to v that avoids w, if such a path exists. We can output this path in $O(|\pi_{sv}|)$ time. Such a data structure (in the static case) was used by Tholey [45] in a linear-time algorithm for the 2-disjoint paths problem in a directed acyclic graph (DAG).

- A decremental version of the fault-tolerant reachability problem [5, 6] in DAGs. We maintain an acyclic flow graph $G = (V, E, s)$ through a sequence of edge deletions, so that we can answer the following query in $O(n)$ time. Given a spanning forest $F = (V, E_F)$ of G rooted at s, find a set of edges $E' \subseteq E \setminus E_F$ of minimum cardinality, such that the subgraph $G' = (V, E_F \cup E', s)$ of G has the same dominators as G.

An incremental low-high order algorithm with $O(mn)$ total update time was presented in [19]. As in the dynamic dominators problem, the decremental version seems more difficult than the incremental. To highlight this aspect, note that a single edge deletion can cause $O(n)$ changes in a given low-high order even if the dominator tree remains unaltered. See Figure 2. On the other hand, in the incremental setting, it suffices to update the low-high order only for the vertices that change parent in the dominator tree.

Our third contribution is an efficient fully dynamic algorithm for maintaining the dominator tree of a DAG under an intermixed sequence of edge insertions and deletions. We obtain this algorithm by incorporating the insertion method of [23] in our decremental algorithm. The fully dynamic algorithm does not preserve the $O(mn)$ worst case bound of the decremental algorithm because the vertex depths in the dominator tree no longer change monotonically. Despite this, however, our experimental results show that it performs very well in practice.

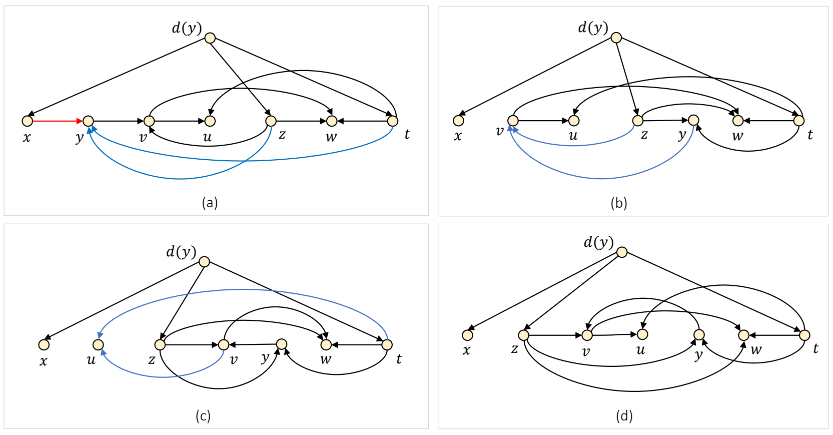

Figure 2 An example of propagation of changes in the low-high order after the deletion of an edge. Vertices are arranged from left to right in low-high order. (a) After the deletion of (x, y), y violates the given low-high order. (b)-(c) Moving y between z and t causes a new violation at vertex v, which in turn causes another violation at vertex u after v is placed between z and y. (d) The low-high order is finally restored when we place u between v and t.

2 Preliminaries

Let $G = (V, E, s)$ be a flow graph with start vertex s, and let D be the dominator tree of G. For any vertex $v \in V$, we let $In(v)$ denote the set of vertices that have an edge in G entering v, i.e., $In(v) = \{u \in V : (u, v) \in E\}$. An edge (x, y) of flow graph G is a *bridge* if its deletion makes y unreachable from s. Given a rooted tree T, we denote by $T(v)$ the subtree of T rooted at v (we also view $T(v)$ as the set of descendants of v). Let T be a tree rooted at s

with vertex set $V_T \subseteq V$, and let $t(v)$ denote the parent of a vertex $v \in V_T$ in T. If v is an ancestor of w, $T[v, w]$ is the path in T from v to w. In particular, $D[s, v]$ consists of the vertices that dominate v. If v is a proper ancestor of w, $T(v, w]$ is the path to w from the child of v that is an ancestor of w. Analogously, $T[v, w)$ denotes the path from v to $t(w)$. Suppose now that the vertex set V_T of T consists of the vertices reachable from s. Tree T has the *parent property* if for all $(v, w) \in E$ with v and w reachable, v is a descendant of $t(w)$ in T. If T has the parent property and has a low-high order, then $T = D$ [25]. For any vertex $v \in V$, we denote by $C(v)$ the set of children of v in D. A *preorder* of T is a total order of the vertices of T such that, for every vertex v, the descendants of v are ordered consecutively, with v first. A preorder of D is a low-high order of G, if $(d(v), v) \in E$ or there are two edges $(u, v), (w, v) \in E$ such that $u < v < w$, and w is not a descendant of v in D.

A *reducible* flow graph [30, 44] is one in which every strongly connected subgraph S has a single *entry* vertex $v \in S$ such that every path from s to a vertex in S contains v. A flow graph is reducible if and only if it becomes acyclic when every edge (v, w) such that w dominates v is deleted [44]. We refer to such an edge as a *back edge*. Deletion of such edges reduces the problem of computing dominators on a reducible flow graph to the same problem on an acyclic graph.

3 Decremental dominators

The algorithm of Georgiadis et al. [20] is based on the concept of *derived edges*. Recall that from the parent property of D, for any edge (v, w) of G, $d(w)$ is an ancestor of v in D. Let (v, w) be an edge of G, with w not an ancestor of v in D. (Such edges do not exist if G is acyclic.) Then, the *derived edge* of (v, w) is the edge $(\overline{v}, w)$, where $\overline{v} = v$ if $v = d(w)$, $\overline{v}$ is the sibling of w in D that is an ancestor of v if $v \neq d(w)$. If w is an ancestor of v in D, then the derived edge of (v, w) is null. Note that a derived edge $(\overline{v}, w)$ may not be an original edge of G. Given the dominator tree D of a flow graph $G = (V, E, s)$ and a list of edges $S \subseteq E$, we can compute the derived edges of S in $O(|V| + |S|)$ time [25].

Now consider the effect of an edge deletion on the dominator tree D. Let (x, y) be the deleted edge. We call the deletion of (x, y) *regular* if (x, y) is not a bridge of G, i.e., y remains reachable from s after the deletion. We let G' and D' denote the flow graph and its dominator tree after the update ($G' = G \setminus (x, y)$). Similarly, for any function f on V, we let f' be the function after the update. In particular, $d'(v)$ denotes the parent of v in D'. By definition, $D' \neq D$ only if x is reachable before the update. We say that a vertex v is *affected* by the update if $d'(v) \neq d(v)$, and *unaffected* otherwise. If v is affected then $d'(v)$ does not dominate v in G. Since the effect of an edge deletion is the reverse of an edge insertion, [23, Lemma 1], and [25, Lemma 4.1] imply the following:

▶ **Lemma 1.** *Suppose x is reachable and the deletion of edge (x, y) is regular, i.e., y does not become unreachable after the deletion. Then the following statements hold:*

(a) *All affected vertices become descendants in D' of a child c of $d(y)$.*

(b) *A vertex v is affected if and only if $(d(v), v)$ is not an edge of G' and all edges $(u, v) \in E \setminus (x, y)$ correspond to the same derived edge $(\overline{u}, v) = (c, v)$ of G.*

(c) *After the deletion, each affected vertex v becomes a child of a vertex on the critical path $D'[c, d'(y)]$.*

(d) *No vertex on $D'[c, d'(y)]$ is affected. Hence, $D'[c, d'(y)] = D[c, d'(y)]$.*

We note that statements (a) and (c) hold for arbitrary flow graphs, while (b) and (d) are true only for acyclic (and reducible) flow graphs. The algorithm of [20] applies Lemma 1 in order to locate the affected vertices in some topological order of G as follows. For each vertex

v we maintain a count *InSiblings*(v) which corresponds to the number of distinct siblings w of v such that (w, v) is a derived edge. We also maintain the lists *DerivedOut*(v) of the derived edges (v, u) leaving each vertex v. As we locate each affected vertex, we find its new parent in the dominator tree and update the counts *InSiblings* for the siblings of v. The first step is to update *InSiblings*(y). If *InSiblings*$(y) = 1$, then we compute the nearest common ancestor $z = d'(y)$ in D' of all vertices in *In*(y). In this case, by Lemma 1(c), z is a descendant of a sibling c of y in D. Next, we update the *InSiblings*(v) counts for all $v \in$ *DerivedOut*(y). Specifically, we decrement *InSiblings*(v) if $v \in$ *DerivedOut*(c); if *InSiblings*$(v) = 1$ then we identify v as affected and inserted into a FIFO queue Q. Then we repeat the same process for each vertex extracted from Q. We can locate the new parent $d'(v)$ of each affected vertex v in D' as for y, i.e., by computing the nearest common ancestor in D' of all vertices in *In*(v). This way, however, does not guarantee the desired $O(mn)$ total update time. Therefore, we locate $d'(v)$ by traversing the critical path $D[c, d'(y)]$ in top-down order, until we find a vertex u such that *In*(v) contains a vertex that is not a descendant of u in D'. Then we have $d'(v) = d(u)$. Finally, we can compute the updated *InSiblings* counts and *DerivedOut* lists in a postprocessing step. The analysis in [20] is based on the fact that the affected vertices that remain reachable increase their depth in D.

3.1 Efficient implementation

Providing a practical version and an efficient implementation of the above algorithm turns out to be a very challenging task. In particular, we need to incorporate efficient solutions to the following subproblems: (i) answering ancestor-descendant queries in the dominator tree D that changes dynamically, (ii) maintaining dynamically the derived edges of G, and (iii) handling the deletion of bridges. We note that (i) and (ii) are not needed when we update D incrementally.

Ancestor-descendant queries. Throughout the execution of the deletion sequence, we need to test in $O(1)$ time the ancestor-descendant relation between pairs of vertices in D, in order to locate the new parent of each affected vertex $v \neq y$. To that end, it suffices to recompute a preorder and a postorder numbering of the vertices in D after each update, since this takes $O(n)$ time by simply performing a dfs traversal of D. Then, v is a descendant of u in D if and only if $u \leq v$ in preorder and $v \leq u$ in postorder [43]. Another option is to represent each order (preorder and postorder) with a data structure for the dynamic list order problem [7, 12]. Both methods guarantee the desired $O(mn)$ total update bound, but the use of a dynamic list order data structure gives a much faster implementation in practice.

Here, we also take advantage of the fact that for each affected vertex v we can move the entire subtree of $D(v)$ in the new location in the dynamic lists, rather than inserting the vertices in $D(v)$ one by one. Specifically, we remove the subtree $D(v)$ from its current locations in the two dynamic lists and insert them immediately after $d'(v)$ in the preorder list and immediately before the first descendant of $d'(v)$ in the postorder list. Thus, we have immediate access to the appropriate location in the preorder list, but we still need to find the corresponding location in the postorder list. In order to do this search fast, we maintain, for each $v \in D$, the list $C(v)$ of the children of v in D ordered in preorder. Then, we can find the first descendant of a vertex u in the postorder list by repeatedly following the links to the first child in preorder, until we reach a leaf.

We implemented the dynamic preorder and postorder lists by adapting the dynamic list order data structure of Bender et al. [7] that uses a two-level structure (implementing a numbering scheme) and supports insertions, deletions and order queries in constant amortized time. We extend this structure so that it can also support the following operation:

***move*(u, v, w):** Move the items between u and v (inclusive) from their current location in the dynamic list and insert them right after w.

We implement the above operation as follows. First, we find the representative nodes (in the top-level structure) for u and v. We check if the left-representative (right-representative, respectively) has items in the second-level list that do not belong to the moved set of items; if there are such items then we split the second-level lists and create new representative items. After this step, both representative items and every other representative item between them, have only second-level items that belong to the set we want to move. Hence, we can quickly move the entire set by linking the left-representative and right-representative to their new position in the dynamic list, right after w. Finally, we check if we can merge the representatives that we move or split with their neighbours.

In our experiments, we observed that the above method was several orders of magnitude faster compared to recomputing a preorder and a postorder numbering in D.

Derived edges. Recall that after finding the affected vertices of an edge deletion, we need to compute the updated *InSiblings* counts and *DerivedOut* lists. The only types of edges that may change their corresponding derived edge are (i) edges entering affected vertices, and (ii) edges that enter a former sibling of y from a descendant of an affected vertex. Let S be the set of these edges. As mentioned above, we can compute the derived edges of S in $O(n + |S|)$ time [25], which suffices for our $O(mn)$ bound since every edge in S is adjacent to at least one vertex that changes depth in D. The method given in [25] for computing derived edges is based on bucket sorting using a preorder numbering of D. This is not suitable for our framework, since we do not maintain a preorder numbering of the vertices, but use a dynamic list order data structure instead. Here we propose a more practical method. First we note that for each edge (u, v) of type (ii), i.e., u is a descendant of an affected vertex and $d(v) = d'(v) = d(y)$, we have $\overline{u} = c$. Now let (u, v) be of type (i), i.e., v is affected so $d'(v) \in D'[c, y)$ and u is a descendant of $d'(v)$. If $u = d'(v)$ then $\overline{u} = u$, so suppose u is a proper descendant of $d'(v)$. Let w_v be the next vertex on $D'[c, y]$ following $d'(v)$ ($w_v = d'(d'(v))$), and let z_u be the nearest ancestor of u such that $d'(z_u) \in D'[c, y]$. Then, $\overline{u} = w_v$ if $d'(z_u) \neq d'(v)$, and $\overline{u} = z_u$ if $d'(z_u) = d'(v)$. Note that we have already computed w_v, for each affected vertex v, when we locate its new parent in D'. Hence, it suffices to compute z_u for all edges (u, v) where u is a proper descendant of $d'(v)$. We do that by visiting the ancestors of u until we reach z_u. First we mark all vertices on $D'[c, y]$, so we stop our search when reaching a vertex that has a marked parent. To avoid multiple visits to the same vertices, we maintain at each vertex w a label $l(w)$, initially null. After we locate z_u, we set $l(w) = z_u$ for each visited vertex w. Thus, the next search stops at a vertex w such that $d'(w)$ is marked or $l(w)$ is not null. Therefore, we can compute all the new derived edges in $O(n + |S|)$ time as desired.

Unreachable vertices. After the deletion of an edge (x, y), some vertices may become unreachable. This happens when (x, y) is a bridge of the current flow graph. Since we deal with acyclic graphs, this means that (x, y) is the only edge entering y from a reachable vertex. Hence, we can detect easily if (x, y) is a bridge, since we have *InSiblings*$(y) = 0$ and $d(y) = x$. In order to achieve $O(mn)$ total running time, we can simply recompute the dominator tree from scratch after each such deletion, since the total number of bridges that can appear is at most $n - 1$. In practice, however, this causes a significant slowdown of our algorithm. A better idea is to handle the deletion of a bridge (x, y) as follows:

1. Compute the set of edges Y from vertices in $D(y)$ to vertices in $D \setminus D(y)$. Note that no edge $e \in Y$ is a bridge in $G \setminus (Y \setminus e)$, since for any vertex $v \in D \setminus D(y)$, all edges in $(w, v) \in Y$ correspond to the same derived edge $(\overline{w}, v)$.
2. Process each edge $e \in Y$ as a regular deletion.
3. Delete $D(y)$ from the dominator tree D' of G', and update accordingly the data structures.

Note that Steps 1 and 3 take $O(m)$ time. Also, since in Step 2 we have regular deletions, the total running time remains $O(mn)$.

4 Decremental low-high order

Now we consider how to update a low-high order of an acyclic flow graph $G = (V, E, s)$ after the deletion of an edge (x, y). First, we show how to achieve an $O(mn)$ total update bound using a sparsification technique, similar to the one used for the incremental problem in [19]. The idea is to maintain a subgraph $H = (V, E_H)$ of G with $O(n)$ edges that has the same dominator tree as G. By Lemma 1(c), each vertex v with $(d(v), v) \notin E$ has two entering edges (u, v) and (w, v) such that $\overline{u} \neq \overline{w}$; then, it suffices to add two such edges in H.

▶ **Corollary 2.** *Let $H = (V, E_H)$ be subgraph of an acyclic flow graph G such that E_H contains:*
(a) *All edges $(u, v) \in E$ such that $u = d(v)$.*
(b) *Two edges (u, v) and (w, v) such that $\overline{u} \neq \overline{w}$ for each vertex v with $(d(v), v) \notin E$.*
Then H has the same dominator tree as G. Moreover, a low-high order of H is also a valid low-high order of G.

Note that the two edges in Corollary 2(b) exist by Lemma 1(c). Clearly $H = (V, E_H)$ has $O(n)$ edges as required. Now, we can compute a low-high order of H in $O(|E_H|) = O(n)$ time using the static algorithm of [25]. The algorithm arranges the children $C(x)$ of each non-leaf vertex x of D in a local low-high order δ_x. First, we place all vertices $v \in C(x)$ with $(x, v) \in E$ in arbitrary order in δ_x. Then, we process the remaining children of x in topological order. For each such vertex v, H contains edges (u, v) and (w, v) such that $\overline{u} \neq \overline{w}$, so $\overline{u}$ and $\overline{w}$ precede v in the topological order and are already located in δ_x. Hence, it suffices to insert v in any location in δ_x between $\overline{u}$ and $\overline{w}$. When we have computed all the local low-high orders, we can get the complete low-high order of G by arranging each subtree $D(v)$ of D immediately after v. After the deletion of (x, y) we need to update H in order to ensure that it still satisfies Corollary 2. We can do this during the update of the derived edges, after we have located all their affected vertices and their new parents in D'. Therefore, we get the following result.

▶ **Theorem 3.** *We can maintain a low-high order of a reducible flow graph G with n vertices through a sequence of edge deletions in $O(mn)$ total time, where m is number of edges in G before all deletions.*

4.1 Bounded search algorithm

Here we present an algorithm that updates a low-high order much faster in practice than the above algorithm. To that end, we also need to maintain the lists *DerivedIn*(v) of the derived edges (u, v) entering each vertex v. The algorithm is based on two ideas. First, we observe that it is easy to update the low-high order for the affected vertices. The problematic case is to update the low-high order for unaffected vertices. For the latter case, we propose a bounded search process that identifies the vertices that may need to be relocated in low-high order.

Affected vertices. The crucial observation is that the decremental dominators algorithm of Section 3 discovers the affected vertices in topological order. Thus, after we move all the affected vertices in their new locations in D' and update their incoming derived edges, we can position them in low-high order. For each affected vertex v, if $(d(v), v) \notin E$, then *DerivedIn*(v) contains two vertices u and w such that $u < w$ in low-high order, so we can insert v between these two vertices.

Unaffected vertices. Now we deal with the more challenging case of updating the low-high order of unaffected vertices. As we already observed, a single edge deletion may cause many changes in a given low-high order, even if there are no affected vertices. (See Figure 2.) Our first step is to identify the initial set I of unaffected vertices that violate the low-high order after updating the dominator tree and the low-high order of the affected vertices. Fixing the low-high order of the vertices in I may invalidate the low-high order of other vertices that are reachable from I. So, our next step is to compute a set X $(I \subseteq X)$ of vertices that may need to be relocated in low-high order due to the changes in the low-high order of I. The next lemma determines the location of the vertices in I.

▶ **Lemma 4.** *Let v be an unaffected vertex that violates the given low-high order after updating the dominator tree in response to an edge deletion (i.e., $v \in I$). Then $d'(v) = d(y)$.*

Proof. A vertex v may violate the low-high order only if it has an entering edge (u, v) such that u is a descendant of an affected vertex and the derived edge of (u, v) changes. From the parent property of the dominator tree we have that for all $(v, w) \in E$ with v and w reachable, v is a descendant of $d(w)$ in D. Since, by Lemma 1(c), all affected vertices become descendants of a child c of $d(y)$, the derived edge of (u, v) changes only if v is a child of $d(y)$. Since v is unaffected, $d'(v) = d(v) = d(y)$. ◀

The above lemma also helps us limit our search for candidate vertices that may need to be relocated in the low-high order in response to the update of the position of the vertices in I. Since I consists only of children of $d(y)$, we only need to search among the unaffected children of $d(y)$ that are reachable from I. As we relocate vertices in low-high order, this process may cascade. See Figure 2.

In order to bound the total running time of our algorithm by $O(mn)$, we maintain a sparse spanning subgraph $H = (V, E_H)$ of G with $O(n)$ edges that satisfies Corollary 2, together with the derived edges $\overline{E}_H$ of E_H. We also maintain the invariant that for each vertex v such that $(d(v), v) \notin E$, the two derived edges $(u, v), (w, v) \in \overline{E}_H$ are such that $u < v < w$ in low-high order.

Our algorithm, FixLH(y), computes a set of vertices $X \subseteq C'(d(y))$ that we will need to process in order to ensure that they satisfy a low-high order of G'. Initially, we set $X = I$ and execute a search from each vertex in I in order to discover vertices that may violate the given low-high order due the replacement of the vertices in I. During this search, we would like to avoid any unnecessary propagation of changes in the low-high order. To achieve this, when we process a vertex $u \in X$, we examine its outgoing derived edges in $\overline{E}_H$. For each such edge (u, v) we test if v satisfies the current low-high order without considering the derived edges from X. If this is not the case, then we insert v into X. This bounded search is outlined by Procedure scan. Note that we can only afford to check a constant number k of entries in *DerivedIn*(v) in order to have $O(n)$ running time per deletion. (In our experiments we set $k \leq 3$.) Thus, we get the following result.

▶ **Lemma 5.** *Algorithm FixLH correctly updates the low-high order of the children of $d(y)$ in D' in $O(n)$ time.*

Algorithm 1 FixLH(y).

```
1  $I$ = children of $d(y)$ that violate the low-high order of $G$ after the deletion /*$I \subseteq \{y\}$
     if $y$ is not affected; otherwise, $I$ contains unaffected children of $d(y)$ that
     have an entering edge from a descendant of an affected vertex */
2  initialize $X = I$ /*$X$ will contain the unaffected children of $d(y)$ that need to be
     relocated in low-high order */
3  foreach vertex $u \in I$ do
4      if $u$ not scanned then scan($u$)
5  end
6  Process vertices in $X$ in topological order to place them in low-high order using the
     edges in $\overline{E}_H$
```

Procedure scan(u).

```
1  foreach derived edge $(u, v) \in \overline{E}_H$ do
2      if $v \notin X$ and $(d(v), v) \notin E$ then
3          if $u < v$ in low-high order then
4              examine the first $k = O(1)$ edges in $DerivedIn(v)$ to find a replacement
                 derived edge $e = (w, v)$ with $w \notin X$ and $w < v$ in low-high order
5          end
6          else
7              examine the first $k = O(1)$ edges in $DerivedIn(v)$ to find a replacement
                 derived edge $e = (z, v)$ with $z \notin X$ and $v < z$ in low-high order
8          end
9          if a replacement derived edge $e$ was found then
10             replace $(u, v)$ with $e$ in $\overline{E}_H$
11         end
12         else
13             insert $v$ into $X$
14             scan($v$)
15         end
16     end
17 end
```

Proof. To prove the correctness of algorithm FixLH, first note that it correctly updates the low-high order of all vertices in X. Now we need to argue that the remaining vertices satisfy the updated low-high order. Observe that any vertex v that is visited during the search for X, is not inserted into X only if $(d(v), v) \in \overline{E}_H$ or if both derived edges in $\overline{E}_H$ entering v are not in X. Clearly, the same holds for all vertices that are not visited during this process. Hence, any vertex $v \notin X$ does not violate the computed low-high order before and after relocating the vertices in X.

Now we argue that the algorithm runs in $O(n)$ time. Each vertex v may change its two entering edges in $\overline{E}_H$ at most $O(1)$ times, since we look for replacement edges only in the first $O(1)$ edges in *DerivedIn*(v). Thus, *DerivedIn*(v) will be examined in lines 4 and 7 of Procedure scan a constant number of times in total for each v, so we spend constant time for each vertex. Finally, we need to process the vertices of X in topological order. Note that the vertices may be inserted in X in arbitrary order. We can sort them topologically by computing a topological order of the of subgraph of $\overline{H} = (V, \overline{E}_H)$ that is induced by the vertices of X. Since $\overline{E}_H$ has $O(n)$ edges, this steps also takes $O(n)$ time. ◀

▶ Remark 6. We also test the following, slightly more complicated, variant of Algorithm FixLH. In line 6, where we place each vertex of $u \in X$ in low-high order, we do not use only the derived edges entering u that are contained in $\overline{E}_H$, but also consider a constant number of derived edges entering u contained in $\overline{E} \setminus \overline{E}_H$. Let $\overline{E}(u)$ be the set of derived edges entering u that we consider in order to decide the location of u in the low-high order. Also, let $\overline{V}(u) = \{w \in V : (w, u) \in \overline{E}(u)\}$, i.e., the vertices from which the derived edges in $\overline{E}(u)$ originate. Then, we place u right after the median vertex in $\overline{V}(u)$, sorted with respect to the low-high order. By doing this placement for u, we hope that Procedure scan will have better chances for locating replacement derived edges.

4.2 Implementation issues

We extend our decremental dominators algorithm of Section 3 so that it also maintains a low-high order as described above. The following implementation issues affect the efficiency of our algorithm in practice.

Representation of a low-high order. Since a low-high order is a preorder of D, we can use the same dynamic list order data structure as in Section 3.1. This choice, however, has the serious drawback that we may need to update the data structures for both the preorder and the postorder of D much more often than in Section 3.1. For this reason, we use a separate dynamic list order data structure for the low-high order, which is updated independently of the preorder and the postorder of D.

Unreachable vertices. As in the decremental dominators algorithm of Section 3.1, we have to take special care of how the deletion of a bridge (x, y) is handled. To that end, we first tested the two methods mentioned in Section 3.1: (a) Run a static algorithm to recompute the dominator tree D and a low-high order from scratch, and (b) Process each edge $e = (u, v)$ with u a descendant of y in D and v not a descendant of y in D as a regular deletion (e cannot be a bridge) and update the low-high order after each such deletion. Then delete (x, y), making all descendants of y in D unreachable from s.

Unlike the decremental dominators algorithm, choice (b) here is not always superior to (a) because during the sequence of regular deletions a vertex may be scanned several times when the FixLH process is executed. Hence, we also implemented the following improvement, which updates the low-high order of unaffected vertices after the sequence of regular deletions is processed. Specifically, we first update the dominator tree as in (b) but do not compute the complete low-high order after each regular deletion of an edge $e = (u, v)$. As we process each regular deletion (u, v), we also fix the low-high order of each affected vertex. Let $A^\star$ denote the set of all affected vertices found during all regular deletions. For each edge (w, t) such that w is a descendant of an affected vertex in $A^\star$ we insert t in a list $I^\star$. We compute a set $X^\star$ of vertices which may need their low-high to be updated by executing $scan(v)$, starting from all vertices v in $I^\star$ that have not been scanned yet. Finally, we sort $X^\star$ topologically and update the low-high order of all vertices in $X^\star$.

All of the above three methods are executed in $O(m)$ time per bridge deletion, so they all guarantee the $O(mn)$ total running time. In our experiments, however, the last method turned out to be an order of magnitude faster than (a) and (b).

5 Empirical Analysis

We wrote our implementations in `C++`, using `g++ v.4.6.4` with full optimization (flag `-O3`) to compile the code. We report the running times on a Dell Precision Tower 7820 CTO Base machine running Ubuntu (16.04 LTS), equipped with an Intel Xeon Gold 5118 2.3 GHz processor with 16 MB L3 cache and 192GB DDR4-2400 RAM at 2,666 MHz. We did not use any parallelization, and each algorithm ran on a single core. We report CPU times measured with the `getrusage` function.

Table 1 Graph instances used in the experiments. The original graphs are taken from [35] and [37], and were converted to DAGs by including vertices and edges reachable from the start vertex and deleting depth-first search back edges. The graph categories are: source code (SC), social network (SN), peer to peer network (P2P), web graph (WG), communication network (CN), and product co-purchasing network (PN). The number of vertices n and edges m refer to the produced instances.

Graph Details				
Graph	**Type**	n	m	**Reference**
linux	SC	1524	3687	KONECT [35]
advogato	SN	2320	17809	KONECT [35]
p2p-Gnutella31	P2P	14149	32363	SNAP [37]
Amazon0302	PN	55414	126663	SNAP [37]
Soc-Epinions1	SN	17117	158754	SNAP [37]
web-BerkStan	WG	29145	169870	SNAP [37]
WikiTalk	CN	49430	664139	SNAP [37]
Amazon0601	PN	276049	1259198	SNAP [37]
web-Google	WG	600493	2013471	SNAP [37]

Table 1 shows some statistics about the graphs used in our experimental evaluation. In all test instances we select the first vertex of the graph as the start vertex. (Choosing a random start vertex produces similar results.) We produce decremental instances as follows. The number of edges that will be deleted is controlled by a parameter $p \in [0,1]$. Let m be the initial number of edges in the graph. We create a sequence of deletions by choosing $\lfloor p \cdot m \rfloor$ edges in the original graph uniformly at random. For each graph and each choice of p, we create 10 such random instances using different seeds for the initialization of the random functions, and report the average running times. (For a given input graph, two values $p_1 < p_2$ of p, and a fixed seed, the deletion sequence for p_1 is a subsequence of the deletion sequence for p_2.) The algorithms compute (in static mode) the dominator tree (and a low-high order for the decremental low-high order algorithms) of the original graph and then they run in decremental mode, processing the sequence of deletions. Note that during the execution of the algorithms some vertices may become unreachable, and thus some subsequent deletions may involve a disconnected portion of the graph. These deletions are detected and ignored by all algorithms. For computing dominators in static mode we use the SNCA algorithm from [27], which is a simplified variant of the classic Lengauer-Tarjan algorithm [36] that performs very well in practice. (Algorithm SNCA was generally faster than other well-known algorithms tested in [27].) As an intermediary, this algorithm computes a sparse subgraph H of the input graph G that has the same dominators as G. The indegree of each vertex in H is at most 2, so H has at most $2(n-1)$ edges (the start vertex has zero indegree). For computing a low-high order, we augment this algorithm with the low-high order algorithm for acyclic graphs from [25]. We speedup the computation of a low-high order by using only the edges in H (instead of all the edges of G).

Dominators. We compare our efficient algorithm, Decr of Section 3, with two dynamized versions of SNCA. (We did not consider the algorithm of Cicerone et al. [10] since it requires $O(n^2)$ space, and therefore is impractical for large graphs.) The first algorithm, DSNCA1, first tests if the deleted edge (x, y) belongs to the sparse subgraph H. If not, then the dominator tree is not affected and the algorithm does nothing. Otherwise, it simply runs SNCA from scratch. The second algorithm, DSNCA2, also performs the same test, but if $(x, y) \in H$, then it computes the nearest common ancestor z of x and y in D and runs SNCA only for the subgraph of G induced by $D(z)$.

The average running times are reported in Table 3. Our experimental results show that the new algorithm Decr is much faster than both DSNCA1 and DSNCA2. In particular, Decr is consistently at least two orders of magnitude faster than DSNCA1 and DSNCA2. For several graphs considered in our experiments, a large fraction of the vertices tended to get disconnected from the start vertex after the deletion of about 50% of the edges, and therefore many subsequent deletions are ignored. We also observe that recomputing the dominator tree only for the subgraph induced by $D(z)$ (where z is the nearest common ancestor of x and y in D) does not provide a significant improvement in the running time of DSNCA, and it may even cause slowdown in some instances, due to the overhead of computing the nearest common ancestor of x and y in D.

Fully-dynamic case. We also report some experimental results for our fully dynamic algorithm, that we refer to as Dyn, that maintains the dominator tree of a DAG under a mixed sequence of edge insertions and deletions. We obtain this algorithm by incorporating the insertion method of [23] in our decremental algorithm. Note that, as in our decremental algorithm, we need to maintain the same data structures for the derived edges and for the dynamic preorder and postorder lists. (These data structures are not required in the incremental setting.) The fully dynamic algorithm does not preserve the $O(mn)$ worst case bound of the incremental or the decremental algorithm because the vertex depths in the dominator tree no longer change monotonically. Despite this, however, our experimental results show that it performs very well in practice.

We produce fully dynamic instances as follows. The number of edges that will be inserted and deleted are controlled by parameters $p_i \in [0, 1]$ and $p_d \in [0, 1]$, respectively. Let m be the initial number of edges in the graph. We create a sequence of insertions and deletions by choosing $m_i = \lfloor p_i \cdot m \rfloor$ random edge insertions and $m_d = \lfloor p_d \cdot m \rfloor$ random edge deletions. This means that the flow graph initially has $m' = m - m_i$ edges. For each graph and each choice of p_i, p_d, we create 10 such random instances using different seeds for the initialization of the random functions, and report the average running times. The algorithms compute (in static mode) the dominator tree of the original graph and then they run in fully-dynamic mode, processing the (intermixed) sequence of insertions and deletions. As in the decremental algorithm, we compute the dominator tree in static mode with the SNCA algorithm from [27]. In the fully-dynamic mode, the type of each update operation is chosen uniformly at random, so that there are m_i insertions interspersed with m_d deletions. During this simulation that produces the dynamic graph instance we keep track of the edges currently present in the graph. If the next operation is a deletion then the edge to be deleted is chosen uniformly at random from the edges in the current graph.

Here we only give some preliminary experimental results, presented in Table 2, and defer to the full version of the paper for a more comprehensive experimental study. As in the decremental setting, we observe significant speed ups with respect to DSNCA1 and DSNCA2.

Table 2 Average running times in seconds over 10 random intermixed update sequences of edge insertions and deletions. The suffixes in the graph names correspond to the percentage of inserted edges and deleted edges, respectively.

Graph_insertion_deletion	Fully Dynamic Dominators		
	DSNCA1	DSNCA2	Dyn
soc-Epinions1_10_10	11.720	9.072	0.248
soc-Epinions1_30_40	10.620	8.956	0.228
soc-Epinions1_40_20	10.696	7.396	0.268
Amazon0302_30_40	0.900	1.424	0.104
web-BerkStan_30_40	0.156	0.284	0.080
web-BerkStan_40_20	0.992	1.584	0.072

Low-high order. Here we examine the efficiency of our algorithm Decr-LH, using the slightly more complicated variant for placing of a vertex in low-high order, described in Remark 6. In some instances, this variant performed significantly better than the simple version, as shown in Figure 3. We compare the running time of Decr-LH with a dynamized version of SNCA that also computes a low-high order of an acyclic flow graph. This algorithm, that we refer to as DSNCA-LH, works as follows. It maintains a sparse subgraph $H = (V, E_H)$ of G such that for each $v \neq s$, $(d(v), v) \in E_H$, or E_H contains edges (u, v) and (w, v) with $u < v < w$. When we delete an edge (x, y) we test if this edge belongs to H. If not, then the dominator tree and the low-high order are not affected, so we do nothing. Otherwise, we look into the entering edges of v and try to find a replacement edge for (x, y) so that y satisfies the current low-high order. If this fails, then we compute the dominator tree and the low-high from scratch.

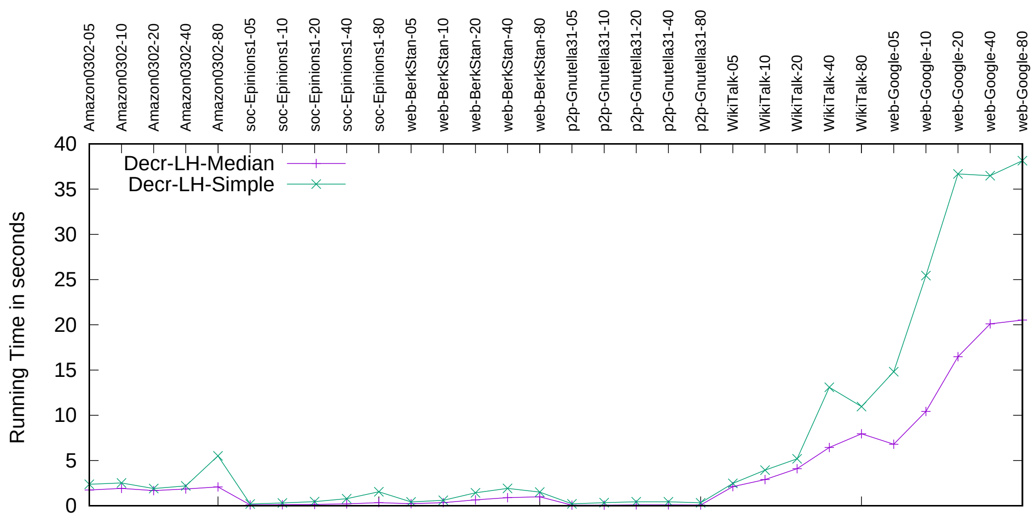

Figure 3 Running time of two implementations of Decr-LH. The first, Decr-LH-Simple, uses the simple method for placing of a vertex in low-high order, as described in Algorithm FixLH, while the second, Decr-LH-Median, uses the method described in Remark 6.

Table 3 Average running times in seconds over 10 random deletion sequences. The suffixes in the graph names correspond to the percentage of deleted edges $p = 5\%, 10\%, 20\%, 40\%$, and 80%. Running times exceeding 2.5 hours are not reported.

Graph_deletion	**Decremental Dominators**			**Decremental Low-High**	
	DSNCA1	**DSNCA2**	**Decr**	**DSCNA-LH**	**Decr-LH**
linux_05	0.060	0.056	**0.001**	0.272	**0.001**
linux_10	0.108	0.116	**0.001**	0.604	**0.001**
linux_20	0.228	0.212	**0.004**	0.974	**0.004**
linux_40	0.428	0.396	**0.004**	1.488	**0.004**
linux_80	0.620	0.640	**0.004**	2.212	**0.004**
advogato_05	0.084	0.084	**0.008**	0.760	**0.008**
advogato_10	0.080	0.080	**0.004**	0.828	**0.012**
advogato_20	0.084	0.080	**0.004**	0.780	**0.008**
advogato_40	0.084	0.080	**0.008**	0.684	**0.008**
advogato_80	0.084	0.080	**0.008**	0.784	**0.008**
p2p-Gnutella31_05	0.916	0.788	**0.020**	4.648	**0.064**
p2p-Gnutella31_10	1.924	1.712	**0.032**	4.912	**0.072**
p2p-Gnutella31_20	2.696	2.520	**0.040**	6.340	**0.100**
p2p-Gnutella31_40	2.260	2.116	**0.048**	5.096	**0.104**
p2p-Gnutella31_80	2.504	2.188	**0.044**	5.368	**0.088**
Amazon0302_05	6.936	10.276	**0.188**	32.136	**1.740**
Amazon0302_10	7.720	11.148	**0.168**	38.216	**1.920**
Amazon0302_20	6.480	10.088	**0.152**	34.404	**1.668**
Amazon0302_40	7.436	11.040	**0.170**	41.552	**1.860**
Amazon0302_80	8.376	11.892	**0.172**	45.224	**2.084**
soc-Epinions1_05	5.116	3.480	**0.072**	10.188	**0.092**
soc-Epinions1_10	10.708	7.684	**0.092**	21.540	**0.112**
soc-Epinions1_20	20.996	14.480	**0.124**	45.308	**0.144**
soc-Epinions1_40	43.012	31.372	**0.204**	97.144	**0.204**
soc-Epinions1_80	100.472	62.528	**0.312**	198.832	**0.340**
web-BerkStan_05	5.204	4.464	**0.060**	16.300	**0.224**
web-BerkStan_10	10.524	8.976	**0.072**	31.724	**0.348**
web-BerkStan_20	18.176	14.468	**0.152**	63.608	**0.656**
web-BerkStan_40	26.252	17.928	**0.204**	118.252	**0.900**
web-BerkStan_80	31.044	23.872	**0.212**	401.040	**0.992**
WikiTalk_05	71.068	97.160	**0.984**	159.068	**2.120**
WikiTalk_10	141.392	192.788	**1.212**	329.100	**2.888**
WikiTalk_20	282.890	386.008	**1.772**	945.708	**4.100**
WikiTalk_40	569.240	746.324	**2.480**	1533.260	**6.436**
WikiTalk_80	923.448	1122.870	**3.364**	2010.560	**7.956**
Amazon0601_05	871.088	790.916	**1.424**	2879.210	**36.380**
Amazon0601_10	1564.340	1417.060	**2.524**	4723.031	**41.568**
Amazon0601_20	2202.820	2118.520	**1.920**	4878.070	**43.888**
Amazon0601_40	2388.700	2068.550	**3.756**	7674.280	**50.004**
Amazon0601_80	2505.030	2395.961	**4.276**	7706.976	**55.644**
web-Google_05	2644.380	>2.5h	**1.320**	11914.512	**6.792**
web-Google_10	4767.340	>2.5h	**1.756**	>2.5h	**10.420**
web-Google_20	7160.680	>2.5h	**3.344**	>2.5h	**16.476**
web-Google_40	>2.5h	>2.5h	**4.444**	>2.5h	**20.108**
web-Google_80	>2.5h	>2.5h	**4.892**	>2.5h	**20.528**

The corresponding average running times are reported in the last two columns of Table 3. As above, our efficient algorithm Decr-LH is much faster than DSNCA-LH. We also observe that in most instances, maintaining a low-high order with our decremental algorithm Decr-LH incurs a relatively low overhead with respect to Decr. For some instances, such as product co-purchasing networks and web graphs (Amazon0302, Amazon0601, web-BerkStan and web-Google), however, the overhead of maintaining a low-high order is significantly higher. In our experiments this was due to the fact that a low-high order may change substantially (i.e., many vertices will be inserted into set X maintained by Algorithm FixLH), even if the dominator tree remains the same. (See Figure 2.)

References

1 S. Allesina and A. Bodini. Who dominates whom in the ecosystem? Energy flow bottlenecks and cascading extinctions. *Journal of Theoretical Biology*, 230(3):351–358, 2004.

2 S. Alstrup, D. Harel, P. W. Lauridsen, and M. Thorup. Dominators in Linear Time. *SIAM Journal on Computing*, 28(6):2117–32, 1999.

3 S. Alstrup and P. W. Lauridsen. A simple dynamic algorithm for maintaining a dominator tree. Technical Report 96-3, Department of Computer Science, University of Copenhagen, 1996.

4 M. E. Amyeen, W. K. Fuchs, I. Pomeranz, and V. Boppana. Fault Equivalence Identification Using Redundancy Information and Static and Dynamic Extraction. In *Proceedings of the 19th IEEE VLSI Test Symposium*, March 2001.

5 S. Baswana, K. Choudhary, and L. Roditty. Fault Tolerant Reachability for Directed Graphs. In Yoram Moses, editor, *Distributed Computing*, volume 9363 of *Lecture Notes in Computer Science*, pages 528–543. Springer Berlin Heidelberg, 2015. doi:10.1007/978-3-662-48653-5_35.

6 S. Baswana, K. Choudhary, and L. Roditty. Fault Tolerant Reachability Subgraph: Generic and Optimal. In *Proc. 48th ACM Symp. on Theory of Computing*, pages 509–518, 2016.

7 M. A. Bender, R. Cole, E. D. Demaine, M. Farach-Colton, and J. Zito. Two Simplified Algorithms for Maintaining Order in a List. In *Proc. 10th European Symposium on Algorithms*, pages 152–164, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg. doi:10.1007/3-540-45749-6_17.

8 A. L. Buchsbaum, L. Georgiadis, H. Kaplan, A. Rogers, R. E. Tarjan, and J. R. Westbrook. Linear-Time Algorithms for Dominators and Other Path-Evaluation Problems. *SIAM Journal on Computing*, 38(4):1533–1573, 2008.

9 M. D. Carroll and B. G. Ryder. Incremental data flow analysis via dominator and attribute update. In *Proc. 15th ACM POPL*, pages 274–284, 1988.

10 S. Cicerone, D. Frigioni, U. Nanni, and F. Pugliese. A uniform approach to semi-dynamic problems on digraphs. *Theor. Comput. Sci.*, 203:69–90, August 1998.

11 R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991. doi:10.1145/115372.115320.

12 P. Dietz and D. Sleator. Two algorithms for maintaining order in a list. In *Proc. 19th ACM Symp. on Theory of Computing*, pages 365–372, 1987.

13 David Eppstein, Zvi Galil, and Giuseppe F. Italiano. Dynamic graph algorithms. In *Algorithms and Theory of Computation Handbook, 2nd Edition, Vol. 1*, pages 9.1–9.28. CRC Press, 2009.

14 W. Fraczak, L. Georgiadis, A. Miller, and R. E. Tarjan. Finding dominators via disjoint set union. *Journal of Discrete Algorithms*, 23:2–20, 2013. doi:10.1016/j.jda.2013.10.003.

15 H. N. Gabow. The Minset-Poset Approach to Representations of Graph Connectivity. *ACM Transactions on Algorithms*, 12(2):24:1–24:73, February 2016. doi:10.1145/2764909.

16 K. Gargi. A Sparse Algorithm for Predicated Global Value Numbering. *SIGPLAN Not.*, 37(5):45–56, May 2002. doi:10.1145/543552.512536.

17 L. Georgiadis. Testing 2-vertex connectivity and computing pairs of vertex-disjoint s-t paths in digraphs. In *Proc. 37th Int'l. Coll. on Automata, Languages, and Programming*, pages 738–749, 2010.

18 L. Georgiadis. Approximating the Smallest 2-Vertex Connected Spanning Subgraph of a Directed Graph. In *Proc. 19th European Symposium on Algorithms*, pages 13–24, 2011.

19 L. Georgiadis, K. Giannis, A. Karanasiou, and L. Laura. Incremental Low-High Orders of Directed Graphs and Applications. In Costas S. Iliopoulos, Solon P. Pissis, Simon J. Puglisi, and Rajeev Raman, editors, *16th International Symposium on Experimental Algorithms (SEA 2017)*, volume 75 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 27:1–27:21, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. `doi:10.4230/LIPIcs.SEA.2017.27`.

20 L. Georgiadis, T. D. Hansen, G. F. Italiano, S. Krinninger, and N. Parotsidis. Decremental Data Structures for Connectivity and Dominators in Directed Graphs. In *ICALP*, pages 42:1–42:15, 2017.

21 L. Georgiadis, G. F. Italiano, L. Laura, and N. Parotsidis. 2-Edge Connectivity in Directed Graphs. In *Proc. 26th ACM-SIAM Symp. on Discrete Algorithms*, pages 1988–2005, 2015.

22 L. Georgiadis, G. F. Italiano, L. Laura, and N. Parotsidis. 2-Vertex Connectivity in Directed Graphs. In *Proc. 42nd Int'l. Coll. on Automata, Languages, and Programming*, pages 605–616, 2015.

23 L. Georgiadis, G. F. Italiano, L. Laura, and F. Santaroni. An Experimental Study of Dynamic Dominators. In *Proc. 20th European Symposium on Algorithms*, pages 491–502, 2012. Full version: *CoRR*, `arXiv:1604.02711`.

24 L. Georgiadis, G. F. Italiano, and N. Parotsidis. Incremental 2-Edge-Connectivity in Directed Graphs. In *ICALP*, pages 49:1–49:15, 2016.

25 L. Georgiadis and R. E. Tarjan. Dominator Tree Certification and Divergent Spanning Trees. *ACM Transactions on Algorithms*, 12(1):11:1–11:42, November 2015. `doi:10.1145/2764913`.

26 L. Georgiadis and R. E. Tarjan. Addendum to "Dominator Tree Certification and Divergent Spanning Trees". *ACM Transactions on Algorithms*, 12(4):56:1–56:3, August 2016. `doi:10.1145/2928271`.

27 L. Georgiadis, R. E. Tarjan, and R. F. Werneck. Finding Dominators in Practice. *Journal of Graph Algorithms and Applications (JGAA)*, 10(1):69–94, 2006.

28 M. Gomez-Rodriguez and B. Schölkopf. Influence Maximization in Continuous Time Diffusion Networks. In *29th International Conference on Machine Learning (ICML)*, 2012.

29 Andreas D.M. Gunawan, Bhaskar DasGupta, and Louxin Zhang. A decomposition theorem and two algorithms for reticulation-visible networks. *Information and Computation*, 252:161–175, 2017. `doi:10.1016/j.ic.2016.11.001`.

30 M. S. Hecht and J. D. Ullman. Characterizations of Reducible Flow Graphs. *Journal of the ACM*, 21(3):367–375, 1974.

31 M. Henzinger, S. Krinninger, and V. Loitzenbauer. Finding 2-edge and 2-vertex strongly connected components in quadratic time. In *Proc. 42nd Int'l. Coll. on Automata, Languages, and Programming*, pages 713–724, 2015.

32 G. F. Italiano, L. Laura, and F. Santaroni. Finding strong bridges and strong articulation points in linear time. *Theoretical Computer Science*, 447:74–84, 2012. `doi:10.1016/j.tcs.2011.11.011`.

33 R. Jaberi. Computing the 2-blocks of directed graphs. *RAIRO-Theor. Inf. Appl.*, 49(2):93–119, 2015. `doi:10.1051/ita/2015001`.

34 R. Jaberi. On computing the 2-vertex-connected components of directed graphs. *Discrete Applied Mathematics*, 204:164–172, 2016. `doi:10.1016/j.dam.2015.10.001`.

35 Jérôme Kunegis. KONECT: the Koblenz network collection. In *22nd International World Wide Web Conference, WWW '13, Rio de Janeiro, Brazil, May 13-17, 2013, Companion Volume*, pages 1343–1350, 2013. `doi:10.1145/2487788.2488173`.

36 T. Lengauer and R. E. Tarjan. A Fast Algorithm for Finding Dominators in a Flowgraph. *ACM Transactions on Programming Languages and Systems*, 1(1):121–41, 1979.

37 J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

38 E. K. Maxwell, G. Back, and N. Ramakrishnan. Diagnosing memory leaks using graph mining on heap dumps. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '10, pages 115–124, 2010.

39 R. M. McConnell, K. Mehlhorn, S. Näher, and P. Schweitzer. Certifying algorithms. *Computer Science Review*, 5(2):119–161, 2011.

40 L. Quesada, P. Van Roy, Y. Deville, and R. Collet. Using Dominators for Solving Constrained Path Problems. In *Proc. 8th International Conference on Practical Aspects of Declarative Languages*, pages 73–87, 2006.

41 G. Ramalingam and T. Reps. An incremental algorithm for maintaining the dominator tree of a reducible flowgraph. In *POPL*, pages 287–296, 1994.

42 V. C. Sreedhar, G. R. Gao, and Y. Lee. Incremental computation of dominator trees. *ACM Transactions on Programming Languages and Systems*, 19:239–252, 1997.

43 R. E. Tarjan. Finding Dominators in Directed Graphs. *SIAM Journal on Computing*, 3(1):62–89, 1974.

44 R. E. Tarjan. Testing Flow Graph Reducibility. *J. Comput. Syst. Sci.*, 9(3):355–365, 1974. `doi:10.1016/S0022-0000(74)80049-8`.

45 T. Tholey. Linear time algorithms for two disjoint paths problems on directed acyclic graphs. *Theoretical Computer Science*, 465:35–48, 2012. `doi:10.1016/j.tcs.2012.09.025`.

46 J. Zhao and S. Zdancewic. Mechanized Verification of Computing Dominators for Formalizing Compilers. In *Proc. 2nd International Conference on Certified Programs and Proofs*, pages 27–42. Springer, 2012. `doi:10.1007/978-3-642-35308-6_6`.