Linear-Time Algorithms for Dominators and Related Problems

Loukas Georgiadis

A Dissertation

Presented to the Faculty

of Princeton University

in Candidacy for the Degree

of Doctor of Philosophy

Recommended for Acceptance

by the Department of

Computer Science

November, 2005

Abstract

This dissertation deals with several topics related to the problem of finding dominators in flowgraphs. The concept of dominators has applications in various fields, including program optimization, circuit testing and theoretical biology. We are interested both in asymptotically fast algorithms and in algorithms that are practical.

We begin with an experimental study of various algorithms that compute dominators efficiently in practice. We describe two practical algorithms that have been proposed in the related literature: an iterative algorithm initially presented by Allen and Cocke and later refined by Cooper, Harvey and Kennedy, and the well-known algorithm of Lengauer and Tarjan. We discuss how to achieve efficient implementations, and furthermore, introduce a new practical algorithm. We present a thorough empirical analysis using real as well as artificial data.

Then we present a linear-time algorithm for dominators implementable on the pointer machine model of computation. Previously, Alstrup, Harel, Lauridsen and Thorup gave a complicated linear-time algorithm for the random-access model. Buchsbaum, Kaplan, Rogers and Westbrook presented a simpler dominators algorithm, implementable on a pointer machine and claimed linear running time. However, as we show, one of their techniques cannot be applied to the dominators problem and, consequently, their algorithm does not run in linear time. Nonetheless, based on this algorithm, we show how to achieve linear running time on a pointer machine.

Next we address the question of how to verify dominators. We derive a linear-time verification algorithm, which is much simpler than the known algorithms that compute dominators in linear time. Still, this algorithm is non-trivial and we believe it provides some new intuition and ideas towards a simpler dominators algorithm.

Finally we study the relation of dominators to spanning trees. Our central result is a linear-time algorithm that constructs two spanning trees of any input flowgraph G, such that corresponding paths in the two trees satisfy a vertex-disjointness property we call *ancestor-dominance*. This result is related to the concepts of *independent spanning trees* and *directed* st*-numberings*, previously studied by various authors, and implies linear-time algorithms for these constructions.

Acknowledgements

I am grateful to my advisor Bob Tarjan for giving me the opportunity to work on these exciting and challenging problems. This thesis could not have been accomplished without his help and support.

I would like to express my gratitude to my coauthors for their contribution to this work. I am especially thankful to Renato Werneck, with whom it has been a real pleasure to work. Adam Buchsbaum and Haim Kaplan provided many insightful suggestions on the linear-time dominators algorithm. I am very happy with the outcome of this effort, which resulted in a much clearer version and presentation of this algorithm. Spyros Triantafyllis assisted me with the preliminary set-up of the experimental analysis on dominators algorithms and provided ideas for future research on related problems arising from his research on compilers.

I wish to thank my thesis readers Adam Buchsbaum and Bernard Chazelle for their helpful suggestions which improved this text, and David August and Moses Charikar for participating in my thesis committee. Finally, I would like to thank Mitra Kelly and Melissa Lawson for their valuable assistance in various administrative issues.

Funding Acknowledgement: This thesis was partially supported by the Aladdin project, NSF grant CCR-0122581.

Στους γονείς μου, *Θεόδωρο και Γεωργία*.

(To my parents, Theodore and Georgia.)

Contents

List of Figures

List of Tables

Chapter 1

Introduction

All the problems that we consider in this dissertation are centered around the concept of *dominators* in *flowgraphs*. In this short chapter we define the dominators problem, provide motivation for its study, and give an overview of the dominators algorithms that have been proposed in the related literature. Finally, we outline the organization of the results presented in the following chapters.

1.1 Dominators in flowgraphs

A *flowgraph* $G = (V, A, r)$ is a directed graph where every vertex in V is reachable from a distinguished root vertex $r \in V$. A vertex w *dominates* a vertex v if every path from r to v includes w. Let *dom*(v) be the set of the vertices that dominate v. Obviously, r and v, the *trivial dominators* of v, are in *dom*(v). For $v \neq r$, the *immediate dominator* of v, denoted by $d(v)$, is the unique vertex $w \neq v$ that dominates v and is dominated by all the vertices in *dom*$(v) - v$. The *(immediate) dominator tree* is a directed tree I rooted at r that is formed by the arcs $\{(d(v), v) \mid v \in V - r\}$. A vertex w dominates v if and only if w is an ancestor of v in I [ASU86]. Thus I is a compact representation of the dominance relation in G. Certain applications require computing the *postdominators* of G, defined as the dominators in the graph obtained from G by reversing all arc orientations. In this setting, it is assumed that

G contains a sink vertex that is reachable from all the vertices of G.

Throughout this dissertation, m is the number of arcs and n is the number of vertices in G.

1.2 Applications

The dominators problem occurs in several application areas, such as program optimization, code generation, circuit testing and theoretical biology.

Compilers make extensive use of dominance information during program analysis and optimization. Perhaps the best-known application of dominators is natural loop detection, which in turn enables a host of natural loop optimizations [Muc97]. Structural analysis [Sha80] also depends on dominance information. Postdominance information is used in calculating control dependences in program dependence graphs [FOW87]. Dominator trees are used in the computation of dominance frontiers [CFR$^+$91], which are needed for efficiently computing program dependence graphs and static single-assignment forms. A dominator-based scheduling algorithm has also been proposed [SB92].

Apart from its applications in compilation, dominators are also used in VLSI testing for identifying pairs of equivalent line faults in logic circuits [AFPB01].

Theoretical biology is another field where dominator analysis has been applied. Specifically, in [AB04, ABB] dominators are used for the analysis of the extinction of species in trophic models (also called *foodwebs*).

1.3 Algorithms for finding dominators

The problem of finding dominators has been extensively studied. In 1972 Allen and Cocke showed that the dominance relation can be computed iteratively from a set of data-flow equations, where the main operation is computing the intersection of sets of vertices [AC72]. A direct implementation of this method has $O(mn^2)$ worst-case time bound, which is very pessimistic in practical settings. A slightly faster algorithm is obtained by representing each set as an array of bits; then the intersection of two sets is formed by a bitwise AND operation [ASU86]. Assuming that b consecutive bits can be manipulated in constant time, the worst-case bound becomes $O(mn^2/b)$. Purdom and Moore [PM72] gave a straightforward dominators algorithm with complexity $O(mn)$. It consists of performing a search in $G - v$ for each $v \in V$; clearly v dominates all the vertices that become unreachable from r. Improving on previous work by Tarjan [Tar74], Lengauer and Tarjan [LT79] proposed an $O(m \log n)$-time algorithm and a more complicated $O(m\alpha(m,n))$-time version, where $\alpha(m,n)$ is a functional inverse of the Ackermann function and it is extremely slow-growing. The Ackermann function has several definitions which are essentially equivalent (up to some constant factors). In [Tar83], Tarjan defines this function by the following recursive relations:

$$\begin{aligned} A(1,j) &= 2^j \text{ for } j \geq 1, \\ A(i,1) &= A(i-1,2) \text{ for } j \geq 1, \\ A(i,j) &= A(i-1, A(i,j-1)) \text{ for } i,j \geq 2. \end{aligned}$$

The inverse $\alpha(m,n)$ is defined as

$$\alpha(m,n) = \min\{i \geq 1 \mid A(i, \lfloor m/n \rfloor) > \log n\}.$$

For all practical purposes this function is a constant not greater than four. Note that for fixed n, $\alpha(m,n)$ is a decreasing function of m/n. In particular, as shown in [Tar75, Tar83], we have

$$\alpha(m,n) \leq 2 \text{ for } \lfloor m/n \rfloor \geq \log^* n, \tag{1.1}$$

where $\log^* n = \min\{i \mid \log^{(i)} n \leq 1\}$ and $\log^{(i)} n$ is defined by $\log^{(0)} n = n$ and $\log^{(i)} n = \log^{(i-1)} \log n$, $i \geq 1$.

Even though the Lengauer-Tarjan (LT) algorithm is fairly complicated, it runs fast in practice. There are even more complicated truly linear-time algorithms. Alstrup et al. [AHLT99] gave a linear-time algorithm for the random-access model of computation.

A simpler algorithm, also implementable on a pointer machine, was proposed by Buchsbaum et al. [BKRW98b]. This algorithm was later shown to have the same time complexity as LT [GT04], and the corrected truly linear-time version is more complicated and requires a RAM (see the Corrigendum of [BKRW98b]). Finally, in joint work with R. E. Tarjan we gave a linear-time algorithm for the pointer-machine computation model [GT04], which is described in Section 3.3.

1.4 Outline

The topics treated in Chapters 3 to 5 are related but fairly independent. Still they have some dependencies that are indicated here. Chapter 2 involves an experimental study of various algorithms that compute dominators efficiently in practice. We discuss how to achieve efficient implementations of some standard algorithms, and furthermore, introduce a new practical algorithm. We present a thorough empirical analysis of these algorithms using real data from four application areas, as well as some artificial data. Part of this work was completed jointly with R. Werneck, R. E. Tarjan, S. Triantafyllis and D. August, and appeared in [GWT$^+$04]. A revised and extended version appears in [GTW].

In Chapter 3 we present a linear-time algorithm for dominators, implementable on the pointer machine model of computation. We start by exhibiting a problem in the analysis of the dominators algorithm proposed by Buchsbaum et al. [BKRW98a, BKRW98b], and provide a modified algorithm that overcomes this problem and achieves linear running time on a pointer machine. Our algorithm (as well as the Buchsbaum et al. algorithm) is based on the algorithm proposed by Lengauer and Tarjan in [LT79], which is reviewed in Section 2.2. Moreover, we present several variants of our basic algorithm, which provide further intuition on the dominators problem. An extended abstract with an earlier version of the material covered in Sections 3.2.3, 3.3 and 3.6 appeared in [GT04] in joint work with R. E. Tarjan. The linear-time algorithms presented in Sections 3.3 and 3.4 will appear in a joint paper with A. L. Buchsbaum, H. Kaplan, A. Rogers, R. E. Tarjan and J. Westbrook [BGK$^+$].

Next, in Chapter 4, we consider the problem of verifying the dominators of a flowgraph. We derive a linear-time algorithm for this verification problem, which is much simpler than the known linear-time algorithms that compute dominators. Still, this algorithm is non-trivial and provides new intuition and ideas which may lead to a simpler algorithm for computing dominators. This work was done jointly with R. E. Tarjan and is covered in [GT05].

In Chapter 5 we study the relation of dominators to spanning trees. Our central result is a linear-time algorithm that, given any flowgraph G, constructs two spanning trees of G, such that corresponding paths in the two trees satisfy a certain vertex-disjointness property. This result is related to the concepts of *independent spanning trees* and *directed st-numberings* (previously studied by various authors) which apply to a restricted class of flowgraphs. We show that an extension of our spanning trees algorithm constructs a directed st-numbering in linear time. Finally, we generalize the related concepts for any flowgraph and provide corresponding linear-time constructions. This is joint work with R. E. Tarjan. The algorithm of Section 5.4 appeared in [GT05]. A journal publication containing all the results of this chapter is in preparation.

Finally, in Chapter 6, we list a few related open problems. The Appendix contains a brief description of concepts from graph theory and algorithms that we use throughout the text and defines the relevant notation.

Chapter 2

Practical Algorithms for Dominators

Experimental results for the dominators problem appear in [LT79, BKRW98b, CHK]. In [LT79] Lengauer and Tarjan found the almost-linear-time version of their algorithm (LT) to be faster than the simple $O(m \log n)$ version even for small graphs. They also showed that the Purdom-Moore [PM72] algorithm is only competitive for graphs with fewer than 20 vertices, and that a bit-vector implementation of the iterative algorithm, by Aho and Ullman [AU77], is 2.5 times slower than LT for graphs with more than 100 vertices. Buchsbaum et al. [BKRW98b] showed that their original (non-linear-time algorithm) has low constant factors, being only about 10% to 20% slower than LT for graphs with more than 300 vertices. Cooper et al. [CHK] presented clever tree-based space- and time-efficient implementation of the iterative algorithm, which they claimed to be 2.5 times faster than the simple version of LT. However, a more careful implementation of LT later led to different results (personal communication). Still, in most practical settings the Cooper et al. algorithm is competitive with more sophisticated algorithms and has the advantage of being very simple to implement.

In this chapter, we present a thorough experimental analysis of algorithms that compute dominators efficiently in practice. Specifically, we explore the effects of different initializations and processing orderings on the tree-based iterative algorithm. We also discuss implementation issues that make both versions of LT faster in practice and competitive with simpler algorithms even for small graphs. Furthermore, we introduce a new practical algorithm that combines LT with the iterative algorithm. For our empirical analysis we use real as well as artificial data. We have not included linear-time algorithms in this study; they are significantly more complex and thus unlikely to be faster than LT in practice. These algorithms have theoretical value and are discussed in Chapter 3.

2.1 The iterative algorithm

Without loss of generality we can assume that r has no entering arcs, since they have no effect on dominators. Then the sets $\mathit{dom}(v)$ are the unique maximal solution to the following data-flow equations:

$$\mathit{dom}'(v) = \Big(\bigcap_{u \in \mathit{pred}(v)} \mathit{dom}'(u) \Big) \cup \{v\},\ \forall\, v \in V. \tag{2.1}$$

As Allen and Cocke [AC72] showed, one can solve these equations iteratively by initializing

$$\mathit{dom}'(v) \leftarrow \begin{cases} \{r\}\,, & v = r \\ V, & \text{otherwise} \end{cases},$$

and repeatedly applying the following step until it no longer applies:

> Find a vertex v such that (2.1) is false and replace $\mathit{dom}'(v)$ by the expression on the right side of (2.1).

A simple way to perform this iteration is to cycle repeatedly through all the vertices of V until no $\mathit{dom}'(v)$ changes. It is unnecessary to initialize all the sets $\mathit{dom}'(v)$ if uninitialized sets are excluded from the intersection in (2.1) and we assign $\mathit{dom}'(r) \leftarrow \{r\}$. In this case, an iterative step is applied to a vertex v only if a value has been computed for at least one $u \in \mathit{pred}(v)$. It is also possible to initialize more accurately the sets $\mathit{dom}'(v)$. Specifically, if S is any subgraph of G and also a tree (spanning or not) rooted at r, we can initialize $\mathit{dom}'(v)$ for $v \in S$ to be the set of ancestors of v in S, and leave $\mathit{dom}'(v)$ for $v \notin S$ uninitialized.

Cooper et al. [CHK] improved the efficiency of this algorithm by observing that we can represent all the sets $\mathit{dom}'(v)$ by a single tree and perform an iterative step as an update of the tree. Specifically, we begin with any tree T rooted at r and repeat the following step until it no longer applies:

Find a vertex v such that

$$\mathit{pred}(v) \cap T \neq \emptyset \text{ and } p_T(v) \neq \mathrm{NCA}(T, \mathit{pred}(v));$$

replace $p_T(v)$ by $\mathrm{NCA}(T, \mathit{pred}(v))$.

(If v is not currently in T then $p_T(v)$ is defined as *null*.) The correspondence between this algorithm and the original algorithm is that for each vertex in T, $\mathit{dom}'(v)$ is the set of ancestors of v in T. The intersection of $\mathit{dom}'(u)$ and $\mathit{dom}'(v)$ is the set of ancestors of $\mathrm{NCA}(T, \{u, v\})$ in T. Once the iteration stops, the current tree T is the dominator tree I. One can also perform the iteration arc-by-arc rather than vertex-by-vertex, replacing $p_T(v)$ by $\mathrm{NCA}(T, \{p_T(v), u\})$ for an arc (u, v) such that $u \in T$ (Figure 2.1). The most straightforward implementation is to cycle repeatedly through the vertices (or arcs) until T does not change.

The number of iterations through the vertices (or arcs) depends on the order in which the vertices (or arcs) are processed. Kam and Ullman [KU76] show that certain data-flow equations, including (2.1), can be solved in at most $l(G, D)+3$ iterations when the vertices are processed in reverse postorder with respect to a DFS tree D. Here $l(G, D)$ is the *loop connectedness of G with respect to D*, the largest number of back arcs found in any cycle-free path of G.

The running time per iteration is dominated by the time spent on NCA computations. If these are performed naïvely (ascending the tree paths until they meet), then a single iteration takes $O(mn)$ time. Because there may be up to $O(n)$ iterations, the running time is $O(mn^2)$. The iterative algorithm runs much faster in practice, however. Typically

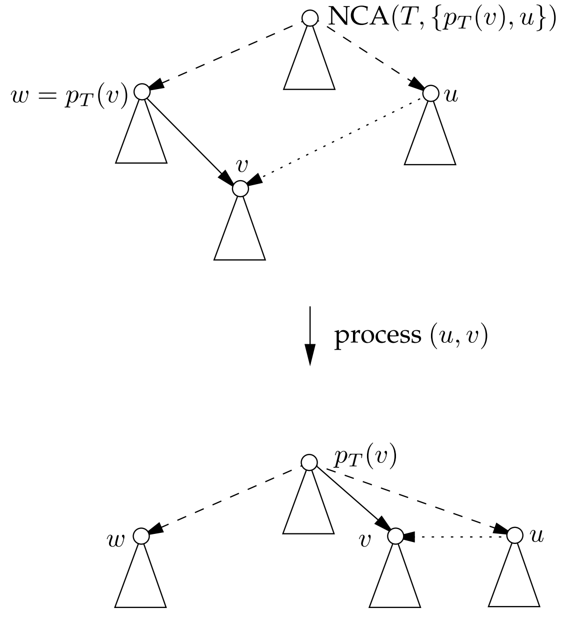


Figure 2.1: The basic step of the iterative algorithm: Processing an arc (u, v). The dashed arcs correspond to tree paths. The dotted arcs are not in T.

$l(G, D) \leq 3$ [Knu71], and it is reasonable to expect that few NCA calculations will require $O(n)$ time. If T is represented as a dynamic tree [ST83], the worst-case bound per iteration is reduced to $O(m \log n)$, but the implementation becomes much more complicated and unlikely to be practical.

2.1.1 Initializations and vertex orderings.

Our base implementation of the iterative algorithm starts with $T \leftarrow \{r\}$ and processes the vertices in reverse postorder with respect to a DFS tree, as done in [CHK]. We refer to this implementation as IDFS (see Figure 2.3). This requires a preprocessing phase that performs a DFS on the graph and assigns a postorder number to each vertex. We do not initialize T as a DFS tree because this is bad both in theory and in practice: it causes the back arcs to be processed in the first iteration, even though they contribute nothing to the NCAs.

Intuitively, a much better initial approximation of the dominator tree is a BFS tree.

We implemented a variant of the iterative algorithm (which we call IBFS) that starts with such a tree and processes the vertices in BFS order. As Section 2.5 shows, this method is often (but not always) faster than IDFS.

We note that there is an ordering σ of the arcs that is optimal with respect to the number of iterations that are needed for convergence. This is stated in the following lemma.

Lemma 2.1 *There exists an ordering σ of the arcs of G such that if the iterative algorithm processes the arcs according to σ, then it will construct the dominator tree of G in a single iteration.*

Proof: We will use the ancestor-dominance spanning trees of Theorem 5.1 (see Chapter 5). This theorem states that G has two spanning trees T_1 and T_2 such that for any v, $T_1[r,v] \cap T_2[r,v] = \mathit{dom}(v)$. We construct σ by catenating a list σ_1 of the arcs of T_1 with a list σ_2 of the arcs of T_2 that are not contained in T_1. The arcs in σ_i are sorted lexicographically in ascending order with respect to a preorder numbering of T_i. The iterative algorithm starts with $T \leftarrow \{r\}$ and after processing the arcs of σ_1 we will have $T = T_1$. We show by induction that after $(p_{T_2}(v), v)$ is processed we will have $p_T(v) = d(v)$. This is immediate for any child of r in T_2. Suppose now that $u = p_{T_2}(v) \neq r$. Since $(p_{T_2}(u), u)$ has been processed before (u, v), we have by the induction hypothesis that $x = \mathrm{NCA}(T, \{u, v\})$ is a dominator of u. Thus, by Theorem 5.1, x is an ancestor of u in both T_1 and T_2. Since $u \in T_2[r,v]$, x is also an ancestor of v in T_2. Note that when a vertex w moves to a new parent in T, it ascends the path $T[r,w]$. So, the set of ancestors of w in T is always a subset of the set of its ancestors in T_1. This implies that x is also an ancestor of v in T_1, and by Theorem 5.1, x dominates v. Now notice that $d(v)$ dominates u. If not, there would be a path P from r to u that avoids $d(v)$, and then P followed by (u, v) would be a path from r to v that avoids $d(v)$, a contradiction. Therefore, $d(v) \in T[r,v] \cap T[r,u]$ and together with the fact $x \in \mathit{dom}(v)$ we conclude that $x = d(v)$. $\square$

When G is acyclic the dominator tree is built in one iteration. This is because reverse postorder is a topological sort of the vertices, so for any vertex v all vertices in $\mathit{pred}(v)$ are processed before v. The iterative algorithm will converge in a single iteration also if G is *reducible* [HU74], i.e., when the repeated application on G of the following operations

(i) delete a loop (v, v);

(ii) if (v, w) is the only arc entering $w \neq r$ delete w and replace each arc (w, x) with (v, x),

yields a single node. Equivalently, G is reducible if every loop has a single entry vertex from r. In a reducible flowgraph, v dominates u whenever (u, v) is a back arc [Tar73]. Therefore, deletion of back arcs, which produces an acyclic graph, does not affect dominators. Alstrup and Lauridsen [AL96] combine this observation together with Gabow's linear-time NCAs algorithm [Gab90] (which allows the input tree to grow by adding leaves), to get a linear-time algorithm for finding dominators in reducible graphs.

Using the algorithm of Section 5.4 we can construct the ancestor-dominance spanning trees, and therefore σ, in almost-linear time, or linear time using the more complicated methods (see Section 3.4). An open question, of both theoretical and practical interest, is whether there is a simple linear-time construction of such an ordering.

One may ask whether any graph has a *fixed* ordering of its vertices that guarantees convergence of the iterative algorithm in a constant number of iterations. The answer to this question is negative; Figure 2.2 shows linearvit(k), a graph family that requires $\Theta(k) = \Theta(n)$ iterations. Note that, if we allow the vertex ordering to be different in each iteration, then Lemma 2.1 trivially implies that two iterations suffice to build the dominators tree.

2.1.2 Marking.

It is reasonable to expect that not all the *dom*′(v) sets will be updated in each iteration. We could obtain significant savings if we could locate and mark the vertices that need to be processed in the next iteration. However, we cannot locate these vertices efficiently, since assigning a new parent in T to a vertex v may require marking all the successors in

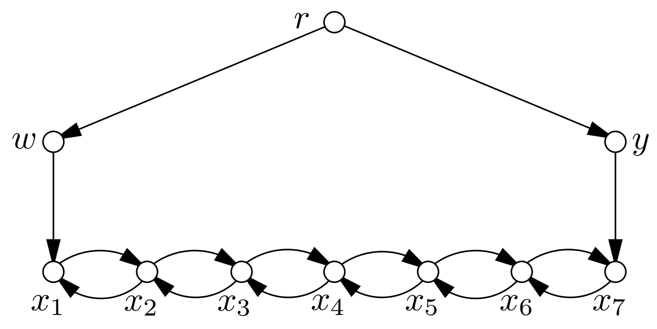


Figure 2.2: Graph family linearvit(k). In this instance $k = 7$. The iterative algorithm needs $\Theta(k)$ iterations to converge when we initialize $T \leftarrow \{r\}$, and the vertices are processed with in any fixed order in each iteration.

G of each vertex in T_v. Indeed, as it turned out in our experiments, this marking scheme added a significant overhead to the iterative algorithm, which became much slower on most graphs (the exceptions were a few artificial graphs). Hence, we did not include these results in Section 2.5.

2.2 The Lengauer-Tarjan algorithm

The Lengauer-Tarjan algorithm initially performs a depth-first search on G starting from r and assigns a preorder number to each vertex it visits. We will refer to the vertices by their preorder number. Let D be the resulting DFS-tree, which we represent by an array *parent*. The algorithm is based on the definition of *semidominators*, which give an initial approximation to the immediate dominators. Lengauer and Tarjan define a path $P = (u = v_0, v_1, \ldots, v_{k-1}, v_k = v)$ in G to be a *semidominator path* (abbreviated as sdom path) if $v_i > v$ for $1 \leq i \leq k-1$. The semidominator of vertex v is defined as

$$s(v) = \min\{u \mid \text{there is an sdom path from } u \text{ to } v\}. \tag{2.2}$$

The next lemma relates the dominators of a vertex with the dominators of its descendants in D.

```
**Algorithm** IDFS($G = (V, A, r)$)

    $D \leftarrow \text{DFS}(r)$, $T \leftarrow \{r\}$, *changed* $\leftarrow$ *true*
    **while** *changed* **do**
        *changed* $\leftarrow$ *false*
        **for** $v \in V - r$ in reverse postorder of $D$ **do**
            $x \leftarrow 0$
            **for** $u \in$ *pred*$(v)$ such that *parent*$[u] \neq 0$ **do**
                **if** *parent*$[v] \neq 0$ **then**
                    **if** $x \neq 0$ **then** $x \leftarrow$ *intersect*$(x, v)$ **else** $x \leftarrow u$ **endif**
                **endif**
            **done**
            **if** $x \neq$ *parent*$[v]$ **then** *parent*$[v] \leftarrow x$, *changed* $\leftarrow$ *true* **endif**
        **done**
    **done**


**int** *intersect*$(x, y)$

    **while** $x \neq y$ **do**
        **while** $x < y$ **do** $x \leftarrow$ *parent*$[x]$ **done**
        **while** $y < x$ **do** $y \leftarrow$ *parent*$[y]$ **done**
    **done**
    **return** $x$
```

Figure 2.3: Algorithm IDFS. The DFS assigns postorder numbers to the vertices. These numbers are used by *intersect* to find NCA$(T, \{x, y\})$. Tree T is represented by an array *parent*. Initially *parent*$[v] = 0$, for all v. During the course of the algorithm we have $v \in T \Leftrightarrow$ *parent*$[v] \neq 0$.

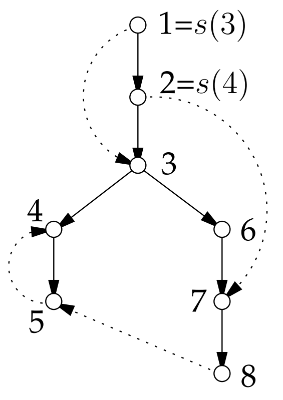


Figure 2.4: Semidominator paths and semidominators. The vertices are numbered in preorder with respect to the DFS tree shown with solid arcs. Non-tree arcs are dotted. The path $(2, 6, 7, 8, 5, 4)$ is an sdom path for 4, hence $s(4) = 2$. The forward arc $(1, 3)$ constitutes an sdom path for 3, so $s(3) = 1$.

Lemma 2.2 [LT79] *Let v and w be any vertices that satisfy $v \overset{*}{\rightarrow} w$. Then $v \overset{*}{\rightarrow} d(w)$ or $d(w) \overset{*}{\rightarrow} d(v)$.*

A relation among $d(w)$, $s(w)$ and w is given in the following result:

Lemma 2.3 [LT79] *For any vertex $w \neq r$, $d(w) \overset{*}{\rightarrow} s(w) \overset{+}{\rightarrow} w$.*

We will use two more lemmas from [LT79]. The first provides an efficient way to compute semidominators and the second suggests how to use semidominators in order to compute immediate dominators.

Lemma 2.4 [LT79] *For any vertex $w \neq r$,*

$$s(w) = \min\big(\{v \mid (v, w) \in A \text{ and } v < w\} \cup \{s(u) \mid u > w \text{ and } \exists\, (v, w) \text{ such that } u \overset{*}{\rightarrow} v\}\big).$$

Lemma 2.5 [LT79] *Let $w \neq r$ and let u be a vertex for which $s(u)$ is minimum among vertices u satisfying $s(w) \overset{+}{\rightarrow} u \overset{*}{\rightarrow} w$. Then $s(u) \leq s(w)$ and $d(u) = d(w)$. Moreover, if $s(u) = s(w)$ then $d(w) = s(w)$.*

The two lemmas imply that we can compute semidominators and immediate dominators by finding minimum semidominator values on paths of D. Moreover, by the properties of DFS it follows that if we process the vertices in reverse preorder then all the necessary values will be available when needed. In order to keep track of the paths on which it evaluates the minima, the algorithm maintains a forest F such that when it needs the minimum $s(u)$ for all u on a path $P = D(w, v]$, then w is the root of the tree in F that contains all the vertices in P. Initially each node in V is a singleton tree in F. For any vertex $v \in V$ we denote by $r_F(v)$ the root of the tree the contains v in F. The operations that are preformed on F are:

link(v): Add arc $(p(v), v)$ to F. This links the tree rooted at v in F to the tree rooted at $p_T(v)$ in F.

eval(v): If $v = r_F(v)$ return v. Otherwise, return a vertex of minimum semidominator among the vertices u that satisfy $r_F(v) \xrightarrow{+} u \xrightarrow{*} v$.

Procedure LT of Figure 2.5 gives the outline of the Lengauer-Tarjan algorithm. We use an array *semi* to store the semidominators of the vertices. After vertex w is processed we have *semi*$[w] = s(w)$. Then w is inserted in a bucket associated with vertex $s(w)$ and is processed again after $s(v)$ has been computed, where v satisfies $s(w) \rightarrow v \xrightarrow{*} w$. At this point we have all the information we need to apply Lemma 2.5 and compute either the immediate dominator or a relative dominator of w. The latter is an ancestor of w that has the same immediate dominator as w. The immediate dominators can be derived from the relative dominators in a simple preorder pass of the vertices (last **for** loop in Figure 2.5).

2.2.1 The simple implementation of *link-eval*.

In order to make evals efficient we use *path compression* in F. Therefore instead of maintaining the forest F we maintain a virtual forest $\mathcal{F}$ with the following properties:

(a) For each T in F there is a corresponding virtual tree $\mathcal{T}$ in $\mathcal{F}$ with the same vertices and the same root as T.

```
**Algorithm** LT($G = (V, A, r)$)
    $D \leftarrow \mathrm{DFS}(r)$
    for $w \in V - r$ in reverse preorder of $D$ do
        $semi[w] \leftarrow w$
        for $v \in pred(w)$ do
            $x \leftarrow eval(v)$
            $semi[w] \leftarrow \min\{semi[w], semi[x]\}$
        done    /* at this point $semi[w] = s(w)$ */
        add $w$ to $bucket[semi[w]]$
        $link(w)$
        $z \leftarrow parent[w]$
        for $v \in bucket[z]$ do
            delete $v$ from $bucket[z]$
            $y \leftarrow eval(v)$
            if $semi[y] < z$ then $idom[v] \leftarrow y$ else $idom[v] \leftarrow z$ endif
        done
    done
    for $w \in V - r$ in preorder of $D$ do
        if $idom[w] \neq semi[w]$ then $idom[w] \leftarrow idom[idom[w]]$ endif
    done
```

Figure 2.5: The Lengauer-Tarjan algorithm.

```
compress($v$)
    $u \leftarrow ancestor[v]$
    if $ancestor[u] \neq 0$ then
        compress($u$)
        if $semi[label[u]] < semi[label[v]]$ then
            $label[v] \leftarrow label[u]$
        endif
        $ancestor[v] \leftarrow ancestor[u]$
    endif
```

```
link($w$)
    $ancestor[w] \leftarrow parent[w]$

int eval($v$)
    if $ancestor[v] \neq 0$ then
        compress($v$)
        return $label[v]$
    else
        return $v$
    endif
```

Figure 2.6: Simple implementation of *link-eval*.

(b) For any vertex v we maintain a value *label*$[v]$ such that

$$semi[eval(v)] = \min\{semi[label[u]] \mid r_{\mathcal{F}}(v) \xrightarrow{+}_{\mathcal{F}} u \xrightarrow{*}_{\mathcal{F}} v\}. \tag{2.3}$$

We represent $\mathcal{F}$ with an array *ancestor*, so that *ancestor*$[v] = p_{\mathcal{F}}(v)$. Path compression is implemented by procedure *compress* of Figure 2.6, which makes every vertex u such that $r_{\mathcal{F}}(v) \xrightarrow{+}_{\mathcal{F}} u \xrightarrow{*} v$ a child of $r_{\mathcal{F}}(v) = r_F(v)$, i.e. we set *ancestor*$[u] \leftarrow r_F(v)$. Initially for all $v \in V$ we have *ancestor*$[v] = 0$. We also use an array *label* to maintain vertices with minimum *semi* values in the compressed paths, so that Property (b) is satisfied. Note that for any vertex v such that *ancestor*$[v] = p_F(v)$ we will have *label*$[v] = v$. This implementation achieves $O(m \log_{(2+m/n)} n)$ running time for $n-1$ *links* and m *evals* [TvL84].

As mentioned in [LT79], we do not in fact need to keep a separate *ancestor* array. Instead we can keep track of the last vertex that was linked, denoted by *lastlinked*. Then we can use the array *parent* to represent both the part of the DFS tree that corresponds to the (unprocessed) vertices $v <$ *lastlinked* and the part of the virtual forrest for the (processed) vertices $v \geq$ *lastlinked*, since these two parts are disjoint. Therefore we can replace the test *ancestor*$[u] \neq 0$ by $u \geq$ *lastlinked*. This modification does not affect later stages of the algorithm since we only need to know which vertex is $p(v)$ when we execute *link*(v). This also implies that in the simple version version of the LT algorithm we don't need to perform *link*(v) explicitly since the linking is implied when we update the value of *lastlinked*. We incorporated these observations in our implementation of the simple version of LT. Henceforth we refer to this implementation as SLT.

2.2.2 The almost-linear-time implementation of *link-eval*.

A more sophisticated linking process, that keeps $\mathcal{F}$ balanced, achieves an $O(m\alpha(m,n))$ running time [Tar79a]. To that end we need to keep for each vertex w a value *size*$[w]$ which is the size of the subtree rooted at w in $\mathcal{F}$. We also require that $\mathcal{F}$ satisfies an additional property:

(c) Each tree $\mathcal{T}$ in $\mathcal{F}$ consists of subtrees ST_i with roots r_i, $0 \leq i \leq k$, such that *semi*[*label*$[r_j]$] $\geq$ *semi*[*label*$[r_{j+1}]$], $0 < j < k$.

Note that ST_i is different from the subtree of $\mathcal{T}$ rooted at r_i, which we denote by $\mathcal{T}_i$, since it does not include ST_j, for $j > i$. Thus, $|ST_i| =$ *size*$[r_i] -$ *size*$[r_{i+1}]$. When we perform *link*(w) the semidominator of $p(w)$ is not known, and we do not require that *semi*[*label*$[r_0]$] $\geq$ *semi*[*label*$[r_1]$]. Note that *eval*(v) for $v \in ST_j$, $j > 0$, does not need the *semi* values on the path $\mathcal{F}[r_0, r_j]$, and therefore it suffices to compress the path $\mathcal{F}[r_j, v]$. We achieve this effect by setting *ancestor*$[r_j] \leftarrow 0$, $0 \leq j \leq k$. Consequently we only need to keep the subtrees ST_i balanced. We keep track of the ST_is using an array *child* such that *child*$[r_i] = r_{i+1}$, $0 \leq i < k$. Note that we need to maintain these values only for subtree roots.

Figure 2.7 gives the code for the almost-linear-time versions of link and eval. When *link*(w) is called we have *semi*$[w] = s(w)$ and Property (c) may not hold. The property is restored by procedure *update*. Also note that we need to modify the *eval* function slightly to account for the fact that after *compress*(v) we may have *ancestor*$[v] \neq r_{\mathcal{F}}(v)$, which means that *label*[*ancestor*$[v]$] could be the required value.

```
int $update(w)$
    $s \leftarrow w$
    $t \leftarrow child[s]$
    while $semi[label[w]] < semi[label[t]]$ do
        if $size[s] + size[child[t]] \geq 2size[t]$ then
            $ancestor[t] \leftarrow s$
            $child[s] \leftarrow child[t]$
        else
            $size[t] \leftarrow size[s]$
            $ancestor[s] \leftarrow t$
            $s \leftarrow ancestor[s]$
        endif
        $t \leftarrow child[s]$
    done
    $label[s] \leftarrow label[w]$
    return $s$
```

```
$link(w)$
    $s \leftarrow update(w)$
    $v \leftarrow parent[w]$
    $size[v] \leftarrow size[v] + size[w]$
    if $size[v] < 2sizew$ then
        $t \leftarrow child[v]$
        $child[v] \leftarrow s$, $s \leftarrow t$
    endif
    while $s \neq 0$ do
        $ancestor[s] \leftarrow v$, $s \leftarrow child[s]$
    done
```

```
int $eval(v)$
    $z \leftarrow ancestor[v]$
    if $z \neq 0$ then
        $compress(v)$
        if $semi[label[z]] \geq semi[label[v]]$
            then return $label[v]$
        else return $label[z]$
        endif
    else return $label[v]$
    endif
```

Figure 2.7: The almost-linear-time implementation of *link-eval*.

Unlike SLT here we can't combine the arrays *parent* and *ancestor*, since we may have *ancestor*$[v] = 0$ even after v is linked. But since *child*$[v] \neq 0$ only if *ancestor*$[v] = 0$ we can combine the arrays *child* and *ancestor* in one array *anchd*. Because we need to distinguish whether v is a (sub)tree root we represent *child* with negative integers. That is, v is a (sub)tree root if and only if *anchd*$[v] \leq 0$. Also *anchd*$[v] > 0$ implies *ancestor*$[v] =$ *anchd*$[v]$ and *anchd*$[v] < 0$ implies *child*$[v] = -$*anchd*$[v]$. We refer to this version of the almost-linear-time LT as LT.

2.2.3 Implementation issues.

Buckets in the Lengauer-Tarjan algorithm have very specific properties:

(1) every vertex is inserted into at most one bucket;

(2) there is exactly one bucket associated with each vertex;

(3) vertex i can only be inserted into some bucket after bucket i itself is processed.

Properties (1) and (2) ensure that buckets can be implemented with two n-sized arrays, *first* and *next*: *first*$[i]$ represents the first element in bucket i, and *next*$[v]$ is the element that succeeds v in the bucket it belongs to. Property (3) ensures that these two arrays can actually be combined into a single array *bucket*.

In [LT79], Lengauer and Tarjan process *bucket*[*parent*$[w]$] at the end of the iteration that deals with w, hence the same bucket may be processed several times. A better alternative is to process *bucket*$[w]$ in the beginning of the iteration; each bucket is now processed exactly once, so it need not be emptied explicitly. Another measure that is relevant in practice is to avoid unnecessary bucket insertions: a vertex w for which *parent*$[w] =$ *semi*$[w]$ is not inserted into any bucket because we already know that $d(w) = p(w)$.

2.3 The SEMI-NCA algorithm

In this section, we introduce SEMI-NCA, a new hybrid algorithm for computing dominators that works in two phases:

(1) Compute $s(v)$ for all $v \neq r$, as done by LT.

(2) Build I incrementally as follows: Process the vertices in preorder. For each vertex w, we ascend the path $I[r, p(w)]$ until we meet the first vertex x such that $x \leq s(w)$, and set x to be the parent of w in I. Here $p(w)$ is the parent of w in the DFS tree used in phase (1).

The correctness of this algorithm is based on the following result:

Lemma 2.6 *For any vertex $w \neq r$, $d(w)$ is the nearest common ancestor in I of $s(w)$ and $p(w)$, i.e.,*

$$d(w) = \mathrm{NCA}(I, \{p(w), s(w)\}).$$

Proof: By Lemma 2.3, we have that $d(w) \xrightarrow{*} s(w) \xrightarrow{*} p(w)$. Obviously, if $p(w) = d(w)$ then $d(w) = s(w)$ and we are done. Now suppose $p(w) \neq d(w)$. Then also $p(w) \neq s(w)$. First we observe that any vertex u in $D(d(w), w]$ is dominated by $d(w)$; if not then there would be a path from r to u that avoids $d(w)$, which catenated with $D[u, w]$ forms a path from r to w that avoids $d(w)$, a contradiction. Hence, both $s(w)$ and $p(w)$ are dominated by $d(w)$.

If $d(w) = s(w)$ then clearly $d(w)$ is the nearest common dominator of $s(w)$ and $p(w)$. Now suppose $d(w) \neq s(w)$. Let v be any vertex such that $d(w) \xrightarrow{+} v \xrightarrow{*} s(w)$. Then there is a path P from $d(w)$ to w that avoids v. Let z be the first vertex on P that satisfies $v \xrightarrow{+} z \xrightarrow{*} p(w)$. This vertex must exist, since otherwise $s(w) < v$ which contradicts the choice of v. Therefore v cannot be a dominator of $p(w)$. Since $d(w)$ dominates both $p(w)$ and $s(w)$, we conclude that $d(w)$ is nearest common ancestor of $s(w)$ and $p(w)$ in I. $\square$

The above lemma implies that $x = d(w)$. Another way to derive this result follows by applying the iterative algorithm on the graph $G' = (V, A', r)$, where A' consists of the arcs $(p(w), w)$ and $(s(w), w)$, for all $w \in V - r$. Clearly the semidominator of any vertex is the same in both G and G'. Hence, by Lemma 2.5, the dominators are also the same. Finally, since G' is acyclic the iterative algorithm IDFS builds the dominator tree in one iteration.

If we perform the search for $d(w)$ naively, by visiting all the vertices in the path $I[d(w), p(w)]$, the second phase runs in $O(n^2)$ worst-case time. However, we expect it to be much faster in practice, since our empirical results indicate that $s(v)$ is usually a good approximation to $d(v)$. SEMI-NCA is simpler than LT in three ways. First, *eval* can return the minimum value itself rather than a vertex that achieves that value. This eliminates one array and one level of indirect addressing (compare *compress* of Figure 2.6 to *snca_compress* of Figure 2.8.) Second, buckets are no longer necessary because the vertices are processed in preorder in the second phase. Finally, it performs one fewer pass over the vertices, since it does not need to compute immediate dominators from relative dominators.

With the simple implementation of *link* and *eval* (which is faster in practice), this method (which we call SNCA) runs in $O(n^2)$ worst-case time. Gabow [Gab90] and later Cole and Hariharan [CH05] showed how to compute NCAs in total linear time on trees that can grow through the addition of leaves.[1] These results imply that the running time of the second phase of SEMI-NCA can be reduced to $O(n)$. In fact, together with Lemma 2.6, these algorithms would yield a linear-time algorithm for computing dominators, if one could compute semidominators in $O(m+n)$ time. However, the algorithms in [Gab90, CH05] are rather complicated, and therefore unlikely to be practical for the computation of dominators.

[1] They also allow the input tree to be modified through other operations.

```
**Algorithm** SNCA$(G = (V, A, r))$
    $D \leftarrow \mathrm{DFS}(r)$
    for $w \in V - r$ in reverse preorder of $D$ do
        for $v \in pred(w)$ do
            $snca\_compress(v)$
            $semi[w] \leftarrow \min\{semi[w], label[v]\}$
        done
        $label[w] \leftarrow semi[w]$
    done
    for $v \in V - r$ in preorder of $D$ do
        while $idom[v] > semi[v]$ do
            $idom[v] \leftarrow idom[idom[v]]$
        done
    done

$snca\_compress(v)$
    $u \leftarrow ancestor[v]$
    if $ancestor[u] \neq 0$ then
        $compress(u)$
        if $label[u] < label[v]$ then $label[v] \leftarrow label[u]$ endif
        $ancestor[v] \leftarrow ancestor[u]$
    endif
```

Figure 2.8: The SEMI-NCA algorithm.

2.4 Worst-case behavior

This section describes families of graphs that elicit the worst-case behavior of the algorithms we implemented. In particular, they show that neither IBFS nor IDFS dominate the other: there are instances on which one algorithm is asymptotically faster than the other. The worst-case graphs also confirm that the time bounds we have presented for SNCA, IBFS, and IDFS are tight. Although such graphs are unlikely to appear in practice, similar patterns may occur within real-world instances. Also, as discussed by Gal et al. [GPF04, GPF05], in an extreme situation a malicious user could exploit the worst-case behavior of optimizing compilers to launch a denial-of-service attack.

Figure 2.9 shows graph families that favor particular methods against the others. For each family, we define a parameter k that controls the size of its members. We denote by $G_k = (V_k, A_k)$ the member that corresponds to a particular value of k.

Iterative. Family itworst(k) contains worst-case inputs for the iterative methods. The set of vertices V_k is defined as $\{r\} \cup \{w_i, x_i, y_i, z_i \mid 1 \le i \le k\}$. The set of arcs A_k is the union of

$$\{(r,w_1),(r,x_1),(r,z_k)\}, \{(w_i,w_{i+1}),(x_i,x_{i+1}),(y_i,y_{i+1}),(z_i,z_{i+1}) \mid 1 \le i < k\},$$

$$\{(z_i,z_{i-1}) \mid 1 < i \le k\}, \{(x_k,y_1),(y_k,z_1)\}, \text{ and } \{(y_i,w_j) \mid 1 \le i,j \le k\}.$$

We have $|V_k| = 4k+1$ and $|A_k| = k^2 + 5k$. Because of the chain of k back arcs (z_i, z_{i-1}), the iterative methods need $\Theta(k)$ iterations to converge. Each iteration requires $\Theta(k^3)$ operations to process the k^2 arcs (y_i, w_j), so the total running time is $\Theta(k^4)$.

Note however that only the dominators of the z_i's change after the first iteration. This fact can be detected by marking the vertices that need to be processed in each iteration, thus processing only the z_i's after the second iteration, and finishing in $\Theta(k^3)$ total time. If we added the arc (z_1, y_k), then all the vertices would have to be marked in each iteration and the total running time would remain $\Theta(k^4)$. As already mentioned, this marking scheme did not work well in our experiments, so we did not include results for it in Section 2.5.

IDFS. Family idfsquad(k) favors IBFS over IDFS. V_k contains the vertices in $\{r\}$ and $\{y_i, x_{i1}, x_{i2} \mid 1 \leq i \leq k\}$. A_k is the union of

$$\{(r, x_1), (r, z_1)\}, \{(x_i, x_{i+1}), (y_i, z_{i+1})\} \mid 1 \leq i < k\} \text{ and}$$

$$\{(x_i, y_i), (y_i, z_i), (z_i, y_i) \mid 1 \leq i \leq k\}.$$

We have $|V_k| = 3k + 1$ and $|A_k| = 5k$. By processing the vertices in reverse postorder, IDFS requires $k + 1$ iterations to propagate the correct dominator values from z_1 to y_k, and the total running time is $\Theta(k^2)$. On the other hand, IBFS processes the vertices in the correct order and constructs the dominator tree in one iteration, and therefore runs in linear time (as do the semidominator-based methods).

IBFS. Family ibfsquad(k) favors IDFS over IBFS. Here V_k is the union of $\{r, w, y, z\}$ and $\{x_i, \mid 1 \leq i \leq k\}$. A_k contains the arcs (r, w), (r, y), (y, z), and (z, x_k), alongside with the sets

$$\{(w, x_i) \mid 1 \leq i \leq k\} \text{ and } \{(x_i, x_{i-1}) \mid 1 < i \leq k\}.$$

Then $|V_k| = k+4$ and $|A_k| = 2k+3$. Processing the vertices in BFS order takes k iterations to reach the fixed point. On the other hand, one iteration with cost $O(k)$ suffices if we order the vertices in reverse postorder, since the graph is acyclic. The semidominator-based methods also run in linear time.

Simple Lengauer-Tarjan. Family sltworst(k) causes worst-case behavior of SLT [Fis72, TvL84]. For any particular k this is a graph with k vertices ($r = x_1, \ldots, x_k$) and $2k - 2$ arcs that causes path compression without balancing to take $\Theta(k \log k)$ time. The graph contains the arcs (x_i, x_{i+1}), $1 \leq i < k$, and $k-1$ arcs (x_i, x_j) where $j < i$, with the property that, after x_j is linked, x_i is a vertex with maximum depth in the tree rooted at x_j of the

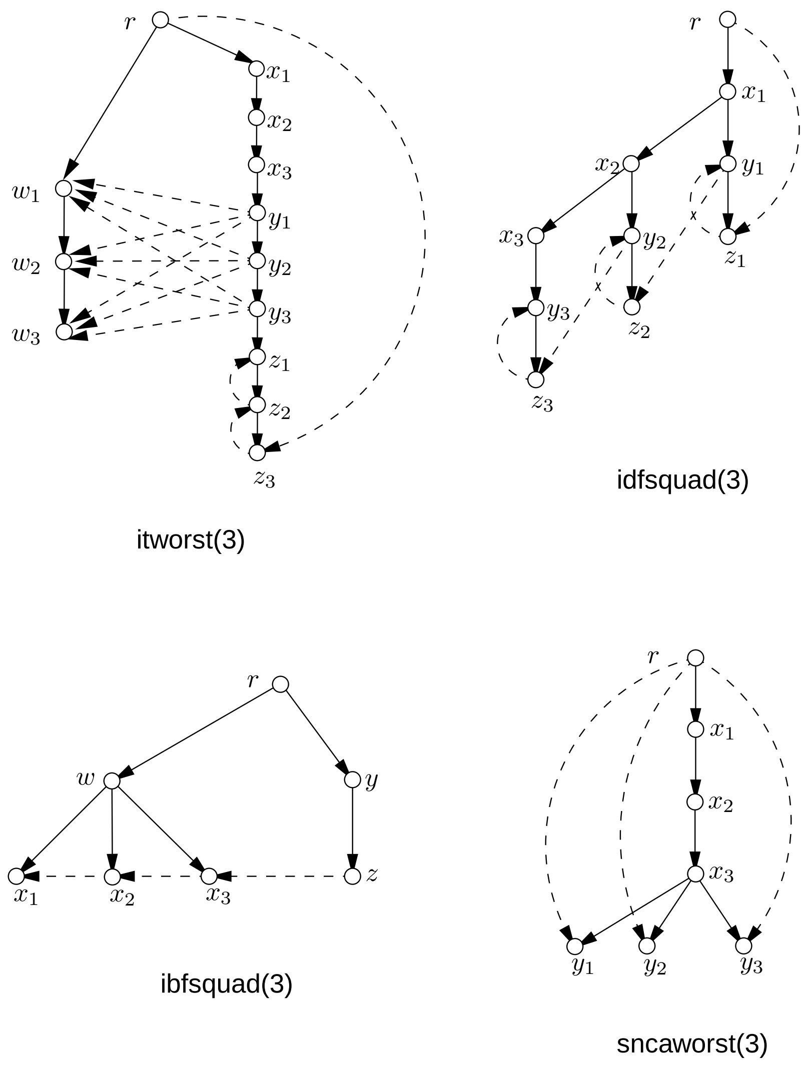


Figure 2.9: Worst-case families for $k = 3$. Subscripts take values in $\{1, \ldots, k\}$. The vertices with no subscript are fixed for each member of a family. The DFS tree used by IDFS, SLT, LT and SNCA is shown with solid arcs; the arcs outside the DFS tree are dashed.

virtual forest $\mathcal{F}$. Note that $d(x_i) = x_{i-1}$ for every $i > 1$. For this reason, the iterative methods need only one iteration to build the dominator tree. However, they still run in quadratic time because they process the same paths repeatedly. It is unclear whether there exists a graph family on which the iterative algorithm runs asymptotically faster than SLT.

SEMI-NCA. Family sncaworst(k) causes the worst-case behavior of SNCA. The set of vertices V_k consists of r, x_i and y_i for $1 \leq i \leq k$. The set of arcs A_k is the union of

$$\{(r, x_1)\}, \{(x_i, x_{i+1}) \mid 1 \leq i < k\} \text{ and } \{(r, y_i), (x_k, y_i) \mid 1 \leq i \leq k\}.$$

We have $|V_k| = 2k + 1$ and $|A_k| = 3k$. Note that $sdom(y_i) = r$ and x_k is the parent of every y_i, which forces SNCA to ascend the path from x_k to r for every y_i. As a result, the algorithm runs in $\Theta(k^2)$ total time. The same bound holds for the iterative methods as well (despite the fact that the graph is reducible), as they also have to traverse the same long path repeatedly. On the other hand, the Lengauer-Tarjan algorithm can handle this situation efficiently because of path compression.

Notice that if we added any arc (y_i, x_k) then $sdom(x_k) = r$, and SNCA would run in linear time. Also BFS would set y_i to be the parent of x_k and r to be the parent of y_i, which implies that IBFS would also run in $\Theta(k)$ time. However, IDFS would still need quadratic time.

2.5 Empirical analysis

Based on worst-case bounds only, the sophisticated version of the Lengauer-Tarjan algorithm is the method of choice among those studied here. In practice, however, "sophisticated" algorithms tend to be harder to code and to have higher constants, so other alternatives might be preferable. The experiments reported in this section shed some light on this issue.

Implementation and experimental setup. We implemented all algorithms in C++. They take as input the graph and its root, and return an n-element array representing immediate dominators. Vertices are assumed to be integers from 1 to n. Within reason, we made all implementations as efficient and uniform as we could. The source code is available upon request.

The code was compiled using `g++` v. 3.3.1 with full optimization (flag `-O4`). All tests were conducted on a Pentium IV with 256 MB of RAM and 256 kB of cache running Mandrake Linux 9.2 at 1.7 GHz. We report CPU times measured with the `getrusage` function. Since its precision is only 1/60 second, we ran each algorithm repeatedly for at least one second; individual times were obtained by dividing the total time by the number of runs. Note that this strategy artificially reduces the number of cache misses for all algorithms, since the graphs are usually small. To minimize fluctuations due to external factors, we used the machine exclusively for tests, took each measurement three times, and picked the best. Running times do not include creating the graph or creating predecessor lists from successor lists (both required by all algorithms). However, times do include allocating and deallocating the arrays used by each method.

Instances. We used control-flow graphs produced by the SUIF compiler [HY97] from benchmarks in the SPEC'95 suite [SPE]. These graphs were previously tested by Buchsbaum et al. [BKRW98b] in the context of dominator analysis. We also used control-flow graphs created by the IMPACT compiler [IMP] from six programs in the SPEC 2000 suite. The instances were divided into *series*, each corresponding to a single benchmark. Series were further grouped into three classes, SUIF-FP, SUIF-INT, and IMPACT. We also considered two variants of IMPACT: class IMPACTP contains the reverse graphs and is meant to test how effectively the algorithms compute postdominators; IMPACTS contains the same instances as IMPACT, with parallel arcs removed. (These arcs appear in optimizing compilers due to superblock formation, and are produced much more often by IMPACT than by SUIF.) We also ran the algorithms on graphs representing circuits from VLSI-testing applications [AFPB01] obtained from the ISCAS′89 suite [CAD] (all 50 graphs were considered a single class), and on graphs representing foodwebs used in [AB04, ABB] (all 21 graphs were considered a single class).

Finally, we tested eight instances that do not occur in any particular application related to dominators. Five are instances from the worst-case families described in Section 2.4, and the other three are large graphs representing speech recognition finite state automata (originally used to test dominator algorithms by Buchsbaum et al. [BKRW98b]).

Test results. We start with the following experiment: read an entire series into memory and compute dominators for each graph in sequence, measuring the total running time. This simulates the behavior of a compiler, which must process several graphs (which typically represent functions) to produce a single executable.

For each series, Table 2.1 shows the total number of graphs (g) and the average number of vertices and arcs (n and m). As a reference, we report the average time (in microseconds) of a simple breadth-first search (BFS) on each graph. Since all the algorithms considered here start by performing a graph search, BFS acts as a lower bound on the actual running time and therefore acts as a baseline for comparison. Times for computing dominators are given as multiples of BFS. Using a baseline method is a standard technique for evaluating algorithms that run in close to linear time in practice [MS94, Gol01a, Gol01b, PW02].

In absolute terms, all algorithms are reasonably fast: none is slower than BFS by a factor of more than six on compiler-generated graphs. The worst relative time observed was slightly more than eight, for FOODWEB. Furthermore, despite their different worst-case complexities, all methods have remarkably similar behavior in practice. In no series was an algorithm twice as fast (or slow) as any other. Differences do exist, of course. LT is consistently slower than SLT, which can be explained by the complex nature of LT and the relatively small size of the instances tested. The iterative methods are usually faster than LT, but often slower than SLT. Both variants (IDFS and IBFS) usually have very

Table 2.1: Complete series: number of graphs (g), average number of vertices (n) and arcs (m), and average time per graph (in microseconds for BFS, and relative to BFS for all dominator algorithms). The best result in each row is marked in bold.

INSTANCE		DIMENSIONS			BFS	RELATIVE TOTAL TIMES				
CLASS	SERIES	g	n	m	TIME	IDFS	IBFS	LT	SLT	SNCA
CIRCUITS	circuits	50	3228.8	5027.2	228.88	5.41	6.35	4.98	3.80	**3.48**
FOODWEB	foodweb	21	78.8	741.0	5.99	6.90	6.95	5.97	4.09	**3.89**
IMPACT	181.mcf	26	26.5	90.3	1.41	4.75	4.36	5.20	3.33	**3.25**
	197.parser	324	16.8	55.7	1.22	4.22	3.66	4.39	3.09	**2.99**
	254.gap	854	25.3	56.2	1.88	3.12	2.88	3.87	2.71	**2.61**
	255.vortex	923	15.1	35.8	1.27	4.04	3.84	4.30	3.24	**3.13**
	256.bzip2	74	22.8	70.3	1.26	4.81	3.97	4.88	3.36	**3.20**
	300.twolf	191	39.5	115.6	2.52	4.58	4.13	5.01	3.51	**3.36**
IMPACTP	181.mcf	26	26.5	90.3	1.41	4.65	4.34	5.09	3.41	**3.21**
	197.parser	324	16.8	55.7	1.23	4.13	3.40	4.21	3.01	**2.94**
	254.gap	854	25.3	56.2	1.82	3.32	3.44	3.79	2.69	**2.68**
	255.vortex	923	15.1	35.8	1.26	4.24	4.03	4.19	**3.32**	**3.32**
	256.bzip2	74	22.8	70.3	1.28	5.03	3.73	4.78	3.23	**3.07**
	300.twolf	191	39.5	115.6	2.52	4.86	4.52	4.88	3.38	**3.33**
IMPACTS	181.mcf	26	26.5	72.4	1.30	4.36	4.04	5.22	3.30	**3.24**
	197.parser	324	16.8	42.1	1.10	4.10	3.56	4.67	3.42	**3.32**
	254.gap	854	25.3	48.8	1.75	3.02	2.82	4.00	2.80	**2.66**
	255.vortex	923	15.1	27.1	1.16	2.59	2.41	3.50	2.45	**2.34**
	256.bzip2	74	22.8	53.9	1.17	4.25	3.53	4.91	3.33	**3.24**
	300.twolf	191	39.5	96.5	2.23	4.50	4.09	5.12	3.50	**3.41**
SUIF-FP	101.tomcatv	1	143.0	192.0	4.23	**3.42**	3.90	5.78	3.67	3.66
	102.swim	7	26.6	34.4	1.04	**2.77**	3.00	4.48	2.97	2.82
	103.su2cor	37	32.3	42.7	1.29	**2.82**	2.99	4.68	3.01	3.03
	104.hydro2d	43	35.3	47.0	1.39	**2.79**	3.05	4.64	2.94	2.86
	107.mgrid	13	27.2	35.4	1.12	**2.58**	3.01	4.25	2.82	2.77
	110.applu	17	62.2	82.8	2.03	**3.28**	3.58	5.36	3.45	3.41
	125.turb3d	24	54.0	73.5	1.51	3.57	3.59	6.31	3.66	**3.44**
	145.fpppp	37	20.3	26.4	0.82	**3.00**	3.43	4.83	3.19	3.19
	146.wave5	110	37.4	50.7	1.43	**3.09**	3.11	5.00	3.22	3.15
SUIF-INT	009.go	372	36.6	52.5	1.72	3.12	3.01	4.71	**3.00**	3.07
	124.m88ksim	256	27.0	38.7	1.17	3.35	**3.10**	4.98	3.16	3.18
	126.gcc	2013	48.3	69.8	2.35	3.00	3.01	4.60	**2.91**	2.99
	129.compress	24	12.6	16.7	0.66	2.79	**2.46**	3.76	2.60	2.55
	130.li	357	9.8	12.8	0.54	2.59	**2.44**	3.92	2.67	2.68
	132.ijpeg	524	14.8	20.1	0.78	2.84	**2.60**	4.35	2.84	2.82
	134.perl	215	66.3	98.2	2.74	3.77	3.76	5.43	**3.44**	3.50
	147.vortex	923	23.7	34.9	1.35	2.69	2.67	3.92	2.59	**2.52**

Table 2.2: Times relative to BFS: geometric mean and geometric standard deviation. The lowest mean in each row is marked in bold.

	IDFS		IBFS		LT		SLT		SNCA	
CLASS	MEAN	DEV	MEAN	DEV	MEAN	DEV	MEAN	DEV	MEAN	DEV
CIRCUITS	5.90	1.18	6.14	1.42	6.74	1.18	4.63	1.15	**4.41**	1.14
FOODWEB	6.67	1.31	6.95	1.28	6.64	1.11	4.16	1.17	**4.04**	1.15
SUIF-FP	2.49	1.44	**2.34**	1.58	3.75	1.42	2.54	1.36	2.96	1.38
SUIF-INT	2.45	1.50	**2.25**	1.62	3.69	1.40	2.48	1.33	2.73	1.45
IMPACT	2.60	1.65	**2.24**	1.76	4.02	1.40	2.74	1.33	2.56	1.31
IMPACTP	2.58	1.63	**2.25**	1.82	3.84	1.44	2.61	1.30	2.52	1.29
IMPACTS	2.42	1.55	**2.05**	1.68	3.62	1.33	2.50	1.28	2.61	1.45

similar behavior, although occasionally one method is much faster than the other (series 145.fppp and 256.bzip2 are good examples). Almost always within a factor of four of BFS (with FOODWEB as the only exception), SNCA and SLT are the most consistently fast methods in the set.

By measuring the total (or average) time per series, the results are naturally biased towards large graphs. For a more complete view, we also computed running times for individual instances, and normalized them with respect to BFS. In other words, for each individual instance we calculated the ratio between the running times of the dominator algorithm and of BFS (the result is the *relative time* of the algorithm). For each class, Table 2.2 shows the geometric mean and the geometric standard deviation of the relative times. Now that each graph is given equal weight, the aggregate measures for iterative methods (IBFS and IDFS) are somewhat better than before, particularly for IMPACT instances. This, together with the fact that their deviations are higher, suggests that iterative methods are faster than semidominator-based methods for small instances, but slower when size increases.

The plot in Figure 2.10 confirms this for the IMPACT class. Each point represents the mean relative running times for all graphs with the same value of $\lceil \log_2(n+m) \rceil$. Iterative methods clearly have a much stronger dependence on size than other algorithms. Almost as fast as a single BFS for very small instances, they become the slowest alternatives as size increases. The relative performance of the other methods is the same regardless of size: SNCA is slightly faster than SLT, and both are significantly faster than LT. A similar behavior was observed for IMPACTS and IMPACTP.

For SUIF, which contains graphs that are somewhat simpler, iterative methods remained competitive even for larger sizes. This is shown in Figure 2.11 for SUIF-INT (the results for SUIF-FP are similar). Note that SLT and SNCA still have better performance as the graph size increases, but now they are closely followed by the iterative method. All algorithms tend to "level-off" with respect to BFS as the size increases, which suggests an almost-linear behavior for this particular class.

Figure 2.12 presents the corresponding results for class CIRCUIT. For the range of sizes shown (note that the graphs are bigger than in the other classes), the average performance of each algorithm (relative to BFS) does not have a strong correlation with graph size.

Finally, Figure 2.13 contains results for the FOODWEB class. On these graphs, which are significantly denser than the others, LT starts to outperform the iterative methods much sooner.

The results for IMPACT and IMPACTS shown in Tables 2.1 and 2.2 indicate that the iterative methods benefit the most by the absence of parallel arcs. Because of path compression, Lengauer-Tarjan and SEMI-NCA can handle repeated arcs in constant time.

So far, we have only compared the algorithms in terms of running times. These can vary significantly depending on the architecture or even the compiler that is used. For a more complete understanding of the relative performance of the algorithms, Table 2.3 shows three architecture-independent pieces of information. The first is SDP, the percentage of vertices (excluding the root) whose semidominators are their parents in the DFS tree. These vertices are not inserted into buckets, so large percentages are better for LT and SLT. On average, far more than half of the vertices have this property. In practice, avoiding unnecessary bucket insertions resulted in a 5% to 10% speedup.

The next two columns show the average number of iterations performed by IDFS

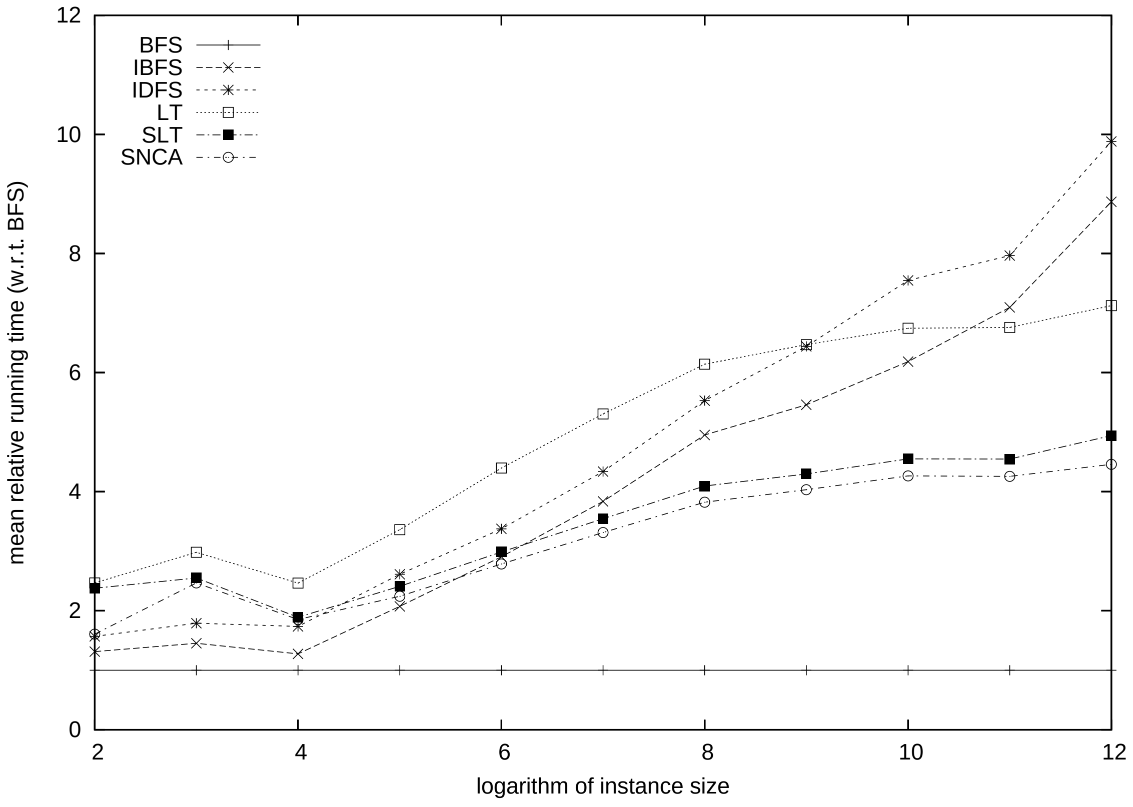


Figure 2.10: Times for IMPACT instances within each size. Each point represents the mean relative running time (w.r.t. BFS) for all instances with the same value of $\lceil \log_2(n + m) \rceil$.

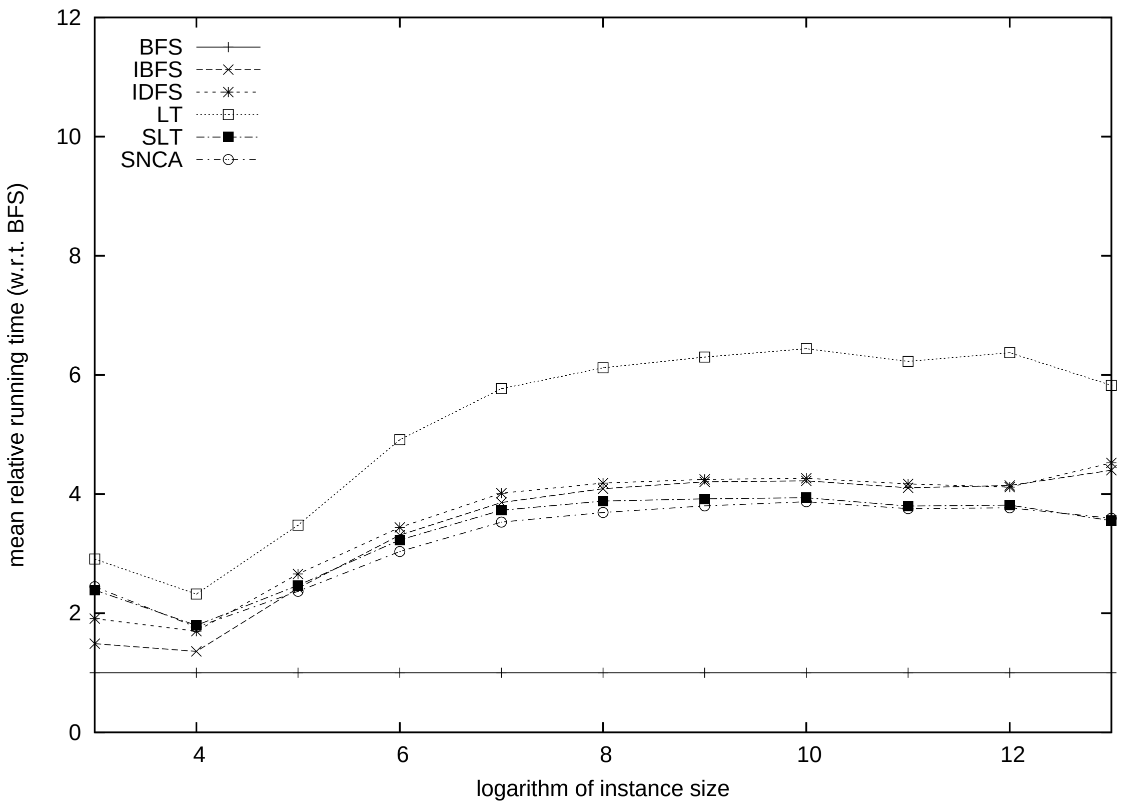


Figure 2.11: Times for SUIF-INT instances within each size. Each point is the mean relative running time (w.r.t. BFS) for instances with the same value of $\lceil \log_2(n+m) \rceil$.

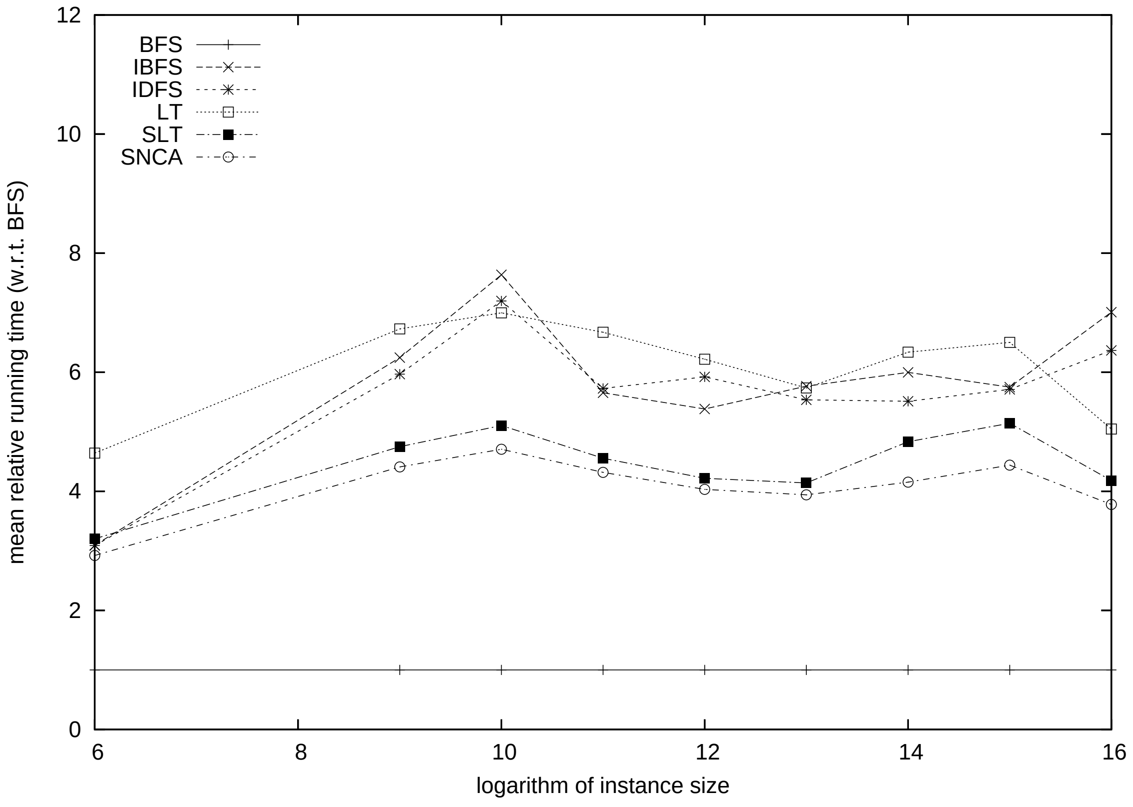


Figure 2.12: Times for CIRCUIT instances within each size. Each point is the mean relative running time (w.r.t. BFS) for instances with the same value of $\lceil \log_2(n+m) \rceil$. Note that the class contains no graph for which this value is 7 or 8.

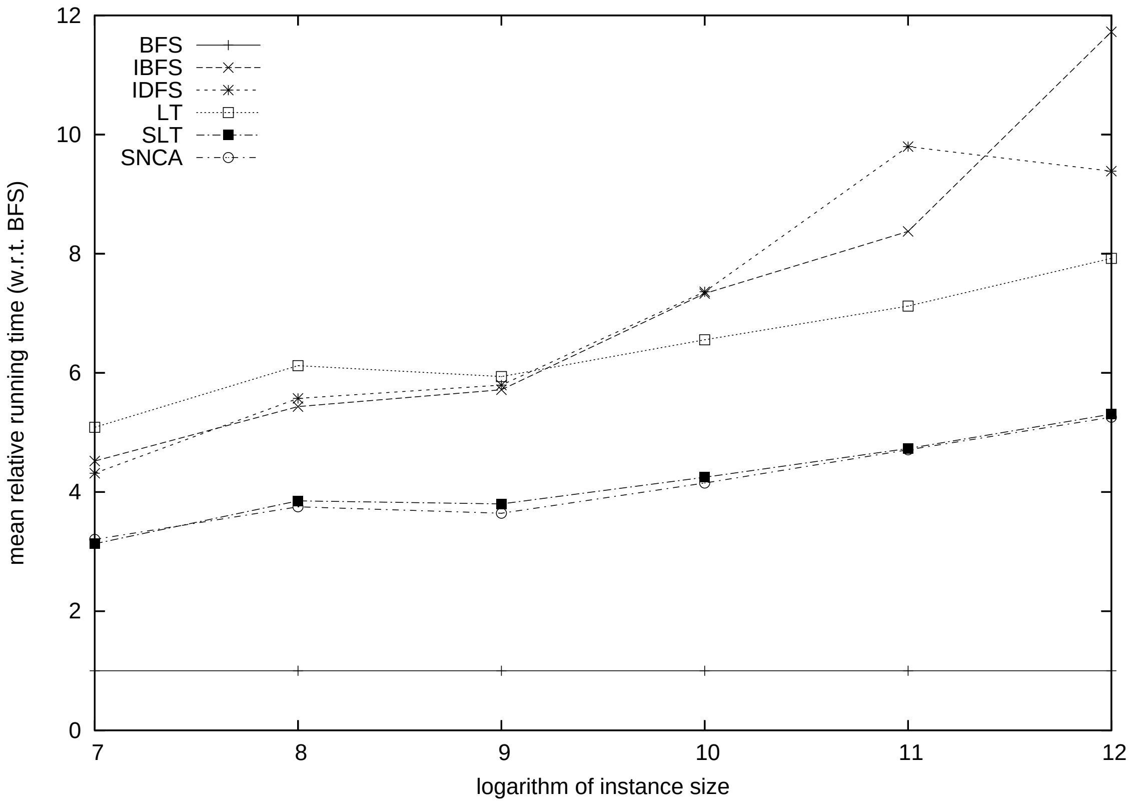


Figure 2.13: Times for FOODWEB instances within each size. Each point is the mean relative running time (w.r.t. BFS) for instances with the same value of $\lceil \log_2(n+m) \rceil$.

Table 2.3: Percentage of vertices that have their parents as semidominators (SDP), average number of iterations, and number of vertex comparisons per arc.

	SDP	ITERATIONS		COMPARISONS PER ARC				
CLASS	(%)	IBFS	IDFS	IBFS	IDFS	LT	SLT	SNCA
CIRCUITS	76.7	3.2000	2.8000	25.3	20.9	7.5	6.2	5.7
FOODWEB	30.9	2.1429	2.1905	12.1	13.0	4.9	4.3	4.3
IMPACT	73.4	1.4385	2.0686	11.1	12.2	6.2	5.1	4.4
IMPACTP	88.6	1.5376	2.0819	12.8	12.0	6.0	4.7	4.3
IMPACTS	73.4	1.4385	2.0686	11.4	12.1	6.8	5.4	4.6
SUIF-FP	67.7	1.6817	2.0000	11.9	9.2	7.5	5.9	5.1
SUIF-INT	63.9	1.6659	2.0009	11.9	10.3	7.6	5.8	5.0

and IBFS. It is very close to 2 for IDFS: almost always the second iteration just confirms that the candidate dominators found in the first are indeed correct. This is expected for control-flow graphs, which are usually reducible in practice. On most classes the average is smaller than 2 for IBFS, indicating that the BFS and dominator trees often coincide. Note that the number of iterations for IMPACTP is slightly higher than for IMPACT, since the reverse of a reducible graph may be irreducible. The small average number of iterations helps explain why iterative algorithms are competitive. In each iteration, they perform one pass over the arcs. In contrast, the other three algorithms perform a single pass over the arcs (to compute semidominators) and one (for SNCA) or two (for SLT and LT) over the vertices.

The last five columns of Table 2.3 show how many comparisons between vertices are performed (normalized by the total number of arcs); the results do not include the initial DFS or BFS. The number of comparisons is always proportional to the total running time; what varies is the constant of proportionality, much smaller for simpler methods than for elaborate ones. Iterative methods need many more comparisons; they are competitive mainly because of smaller constants. In particular, they need to maintain only three arrays, as opposed to six or more for the other methods. (Two of these arrays translate vertex numbers into DFS or BFS labels and vice-versa.)

We end our experimental analysis with results on artificial graphs. For each graph,

Table 2.4: Individual graphs (times for BFS in microseconds, all others relative to BFS). The best result in each row is marked in bold.

INSTANCE			BFS	RELATIVE RUNNING TIMES				
NAME	VERTICES	ARCS	TIME	IDFS	IBFS	LT	SLT	SNCA
itworst	401	10501	34	6410.5	6236.8	9.2	**4.7**	**4.7**
idfsquad	1501	2500	28	2735.3	21.0	8.6	**4.2**	10.5
ibfsquad	5004	10003	88	4.9	9519.4	8.8	4.5	**4.3**
sltworst	32768	65534	2841	283.4	288.6	**7.9**	11.0	10.5
sncaworst	10000	14999	179	523.2	243.8	12.1	**8.3**	360.7
atis	4950	515080	2607	8.3	12.8	6.5	3.5	**3.3**
nab	406555	939984	49048	17.6	15.6	12.8	11.6	**10.2**
pw	330762	823330	42917	18.3	15.1	13.3	12.1	**10.4**

Table 2.4 shows the number of vertices and arcs, the time for BFS (in microseconds), and the times for computing dominators (as multiples of BFS). The first five entries represent the worst-case families described in Section 2.4. In all cases, the algorithms behave as predicted. The speech-recognition graphs (atis, nab and pw) have no special adversarial structure, but are significantly larger than other graphs. As previously observed, the performance of iterative methods tends to degrade more noticeably with size. SNCA and SLT remain the fastest methods, but the asymptotically better behavior of LT starts to show.

2.6 Final remarks

We compared five algorithms for computing dominators. Results on three classes of application graphs (program flow, VLSI circuits, and speech recognition) indicate that they all have similar overall performance in practice. The tree-based iterative algorithms proposed by Cooper et al. are by far the easiest to code and use less memory than the other methods, which makes them perform particularly well on small, simple graphs. For the compiler-generated graphs we tested, the iterative algorithms remained competitive even as the size increased. Given their simplicity, they are a good choice for non-critical applications.

Even on small instances, however, we did not observe the clear superiority of the original tree-based algorithm reported by Cooper et al. (which we call IDFS). Both versions of LT and the hybrid algorithm (SEMI-NCA) are more robust on application graphs, and the advantage increases with graph size or graph complexity. Among these three, the sophisticated version of LT was the slowest, in contrast with the results reported by Lengauer and Tarjan [LT79]. The simple version of LT and hybrid were the most consistently fast algorithms in practice; since the former is less sensitive to pathological instances, we think it should be preferred where performance guarantees are important.

Acknowledgements

We thank Adam Buchsbaum for providing the SUIF and speech recognition graphs, Stefano Allesina and Jennifer Dunne for the foodweb graphs, and Matthew Bridges for his help with the IMPACT compiler.

Chapter 3

Finding Dominators in Linear Time

In this chapter we describe a linear-time algorithm for finding dominators, implementable both on a random-access machine and on a pointer machine. Previously, Harel [Har85] claimed a linear-time algorithm, but it was shown that this algorithm contains several flaws [ALT96]. Based on Harel's approach, Alstrup et al. [ALT96, AHLT99] gave a correct linear-time algorithm on a random-access machine, but in order to achieve this running time their algorithm uses a very complicated data structure, Fredman and Willard's Q-heaps [FW94]. Later, Buchsbaum et al. [BKRW98a, BKRW98b] claimed a "new, simpler" linear-time algorithm with implementations both on a random access machine and on a pointer machine. However, as we show in Section 3.2.3, a key lemma in their analysis does not in fact apply to their dominators algorithm, and the algorithm does not run in linear time. Buchsbaum et al. later gave a fix for their algorithm, which uses a technique implementable on random-access machines but not pointer machines (see Corrigendum in [BKRW98b]). Still, based on the work of Buchsbaum et al., we provide a complete, correct, linear-time dominators algorithm, implementable on either a random-access machine or a pointer machine. One key result is a linear-time reduction of the dominators problem to an off-line nearest common ancestors problem.

Our algorithm is an extension of the Buchsbaum et al. (BKRW) algorithm, which we review in Section 3.2. The BKRW algorithm is based on partitioning a DFS tree D of the input flowgraph G into bottom-level microtrees which participate in a preprocessing phase, and a core tree C on which a modified version of the Lengauer-Tarjan algorithm is applied. The goal of the preprocessing phase is to determine for each vertex v whether $d(v)$ is within the microtree containing v. If this is true then $d(v)$ has been computed during the preprocessing, otherwise it is found during the Lengauer-Tarjan phase. In order to achieve linear running time we extend the BRKW partitioning by dividing C into unary paths. We give the details of this partitioning in Section 3.1. The algorithm proceeds by applying Lengauer-Tarjan computations on a condensed tree that results from C by contracting each unary path to a single vertex. The vertices inside each unary path are processed using stack-based contractions of strongly connected components. We present our algorithm comprehensively in Section 3.3. Sections 3.4, 3.5 and 3.6 present some alternative implementations which rely on various technical observations regarding the operations that are performed by our basic algorithm.

3.1 Partitioning the DFS tree

Let D be any fixed DFS tree of G, and let g be a parameter that we will fix appropriately. We partition D into microtrees of size at most g as follows. If $|D_v| \leq g$ and $|D_{p(v)}| > g$, then D_v is a *non-trivial microtree* with root v. If $|D_v| > g$, then $\{v\}$ itself forms a singleton, *trivial microtree*. Otherwise, v is a non-root vertex in a non-trivial microtree. We denote by *micro*(v) the (trivial or non-trivial) microtree that contains vertex v; we will denote the root of this microtree by *root*(*micro*(v)). The vertices of the trivial microtrees constitute the *core* C of D. For any $v \in D$, let $\eta(v)$ be the nearest ancestor of v in C, that is,

$$\eta(v) = \begin{cases} v, & v \in C \\ p(\mathit{root}(\mathit{micro}(v))), & v \in D - C \end{cases}.$$

We group the vertices of the core into a set of maximal unary paths. For any such path $\ell = (v_1, \ldots, v_k)$ we have that the out-degree of v_i is one for $1 \leq i \leq k-1$ and the out-degree of v_k is zero or greater than one. We call such a path a *line* and denote by

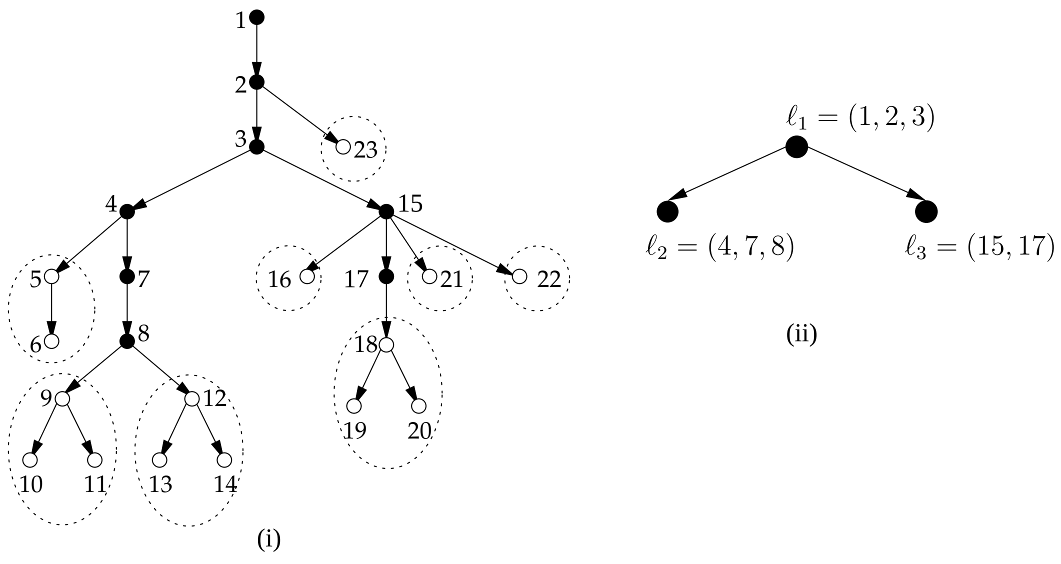


Figure 3.1: Example of the partitioning of a DFS tree D for $g = 3$. Filled nodes are in the core; open nodes are in non-trivial microtrees; (i) D is partitioned to 8 nontrivial microtrees (shown encircled), and 3 lines $\ell_1 = (1,2,3)$, $\ell_2 = (4,7,8)$ and $\ell_3 = (15,17)$. (ii) The compressed core tree C'.

line(v) the line that contains v. Also for any line ℓ we denote by *top*(ℓ) the vertex in ℓ with the lowest DFS number and by *bottom*(ℓ) the vertex in ℓ with the highest DFS number. We call *top*(ℓ) and *bottom*(ℓ) the *endpoints* of ℓ. Note that C has fewer than n/g leaves, since a vertex is a leaf of C if and only if all of its children in D are roots of non-trivial microtrees. Thus, C has $L < 2n/g$ lines. Let C' be the tree that results from contracting each line in C into a single vertex. Then C' has L vertices. For any two vertices $\ell, q \in C'$, ℓ and q correspond to lines in C, and $\ell = p_{C'}(q)$ if and only if *bottom*$(\ell) = p_C($*top*$(q))$. Figure 3.1 gives an example of the partitioning.

Now consider a line $\ell = (v_1, \ldots, v_k)$. The subtree D_{v_i} of D rooted at any v_i is a union of non-trivial microtrees and a part of the core. For each vertex v_i, $1 \leq i \leq k-1$, we define *TR*(v_i) to be the set of microtrees to the *right* of v_i and *TL*(v_i) to be the set of microtrees to the *left* of v_i. More precisely *TR*(v_i) contains all proper descendants w of v_i such that $w > v_{i+1}$ and w is not a descendant of v_{i+1}; *TL*(v_i) contains all proper descendants w of v_i such that $w < v_{i+1}$. *TR*(v_i), *TL*(v_i), and $D_{v_{i+1}}$ partition the proper descendants of v_i. All the children of v_i except for v_{i+1} are roots of non-trivial microtrees. We also define $\mathit{TR}(\ell) = \bigcup_{i=1}^{k-1} \mathit{TR}(v_i)$, which we call the set of microtrees to the *right* of ℓ, and $\mathit{TL}(\ell) = \bigcup_{i=1}^{k-1} \mathit{TL}(v_i)$, which we call the set of microtrees to the *left* of ℓ. The proper descendents of v_k are said to be *below* ℓ.

Procedure *partition* implements the partitioning of the DFS tree D (see Figure 3.2). It begins by calling a procedure that carries out the DFS and builds D. In order to perform the partitioning of G we associate two numbers with each vertex v; *size*$[v]$, which is the size of the subtree rooted at v, and a special vertex defined by

$$\mathit{special}(v) = \begin{cases} \mathit{root}(\mathit{micro}(v)), & v \notin C \\ \mathit{top}(\mathit{line}(v)), & v \in C \end{cases}.$$

To distinguish between v being in the core or not we use a bit *core*$[v]$ which is set to *true* if and only if $v \in C$. Procedure *partition* uses an array *special* to store the special vertices associated with each vertex; *special*$[v]$ is set only when either $v \in C$ and $v = \mathit{top}(\mathit{line}(v))$, or when $v \notin C$ and $v = \mathit{root}(\mathit{micro}(v))$. For any other vertex initially we set *special*$[v] \leftarrow \infty$, and *special*(v) can be computed on demand by walking backwards on the path $D[\mathit{special}(v), v]$ until we find a vertex u with *special*$[u] < \infty$. Then we can set *special*$[w] \leftarrow$ *special*$[u]$ for all the vertices w that we have visited. This is implemented in procedure *find_special*. (A similar procedure is used in [BKRW98b] to locate $\mathit{root}(\mathit{micro}(v))$.) Throughout the course of our algorithm *find_special* will visit a vertex at most twice, therefore spending amortized $O(1)$ time per vertex. We will ignore these details henceforth.

3.2 The BKRW algorithm

3.2.1 External dominators

In this section we review some of the definitions and results given in [BKRW98a, BKRW98b].

A path $P = (u = v_0, v_1, \ldots, v_{k-1}, v_k = v)$ is a *external dominator path* (abbreviated xdom path) if P is an sdom path and $v_i \notin \mathit{micro}(v)$ for $0 \leq i \leq k-1$. The *external*

```
partition$(G = (V, A, r))$
    $D \leftarrow$DFS$(r)$
    for $v$ in reverse preorder do
        for $u \in chd_D(v)$ do
            $size[v] \leftarrow size[v] + size[u]$
        done
        if $size[v] > g$ then
            $core[v] \leftarrow true$
            for $u \in chd_D(v)$ do
                if $size[u] \leq g$ then
                    $special[u] \leftarrow u$
                else
                    $chd_C(v) \leftarrow chd_C(v) \cup \{u\}$
                endif
            done
            if $|chd_C(v)| > 1$ then
                for $u \in chd_C(v)$ do
                    $special[u] \leftarrow u$
                done
            endif
        endif
    done
```

```
int find_special$(v)$
    $u \leftarrow v$
    $P \leftarrow \emptyset$
    while $special[u] = \infty$ do
        $P \leftarrow P \cup \{u\}$
        $u \leftarrow parent[u]$
    done
    for $w \in P$ do
        $special[w] \leftarrow special[u]$
    done
    return $special[v]$
```

Figure 3.2: Procedures that partition D into nontrivial microtrees and lines, and that locate the special vertices.

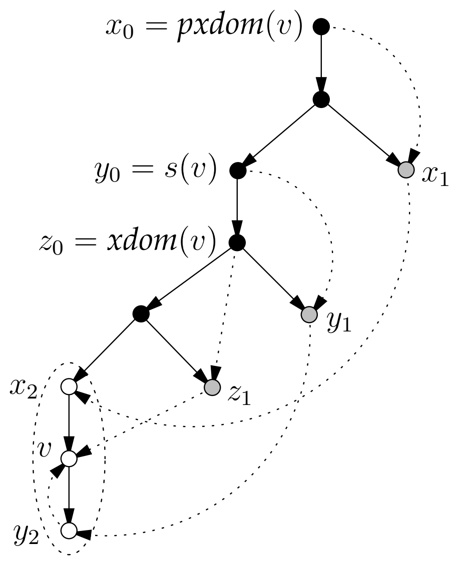


Figure 3.3: Semidominator, external dominator, and pushed external dominator of vertex v, where *micro*(v) is nontrivial. The straight arcs are DFS-tree arcs; non-tree arcs are dotted. Filled nodes are in the core; open nodes are in non-trivial microtrees; gray nodes can be in either state. (x_0, x_1, x_2, v) is a pxdom path; (y_0, y_1, y_2, v) is an sdom path; (z_0, z_1, v) is an xdom path.

dominator of vertex v is defined as

$$\mathit{xdom}(v) = \min\{v\} \cup \{u \mid \text{there is an xdom path from } u \text{ to } v\}.$$

A path $P = (u = v_0, v_1, \ldots, v_{k-1}, v_k = v)$ is a *pushed external dominator path* (pxdom path) if $v_i \geq \mathit{root}(\mathit{micro}(v))$ for $1 \leq i \leq k-1$. The *pushed external dominator* of vertex v is defined as

$$\mathit{pxdom}(v) = \min\{u \mid \text{there is a pxdom path from } u \text{ to } v\}.$$

From [BKRW98b] we have that $d(v) \notin \mathit{micro}(v)$ implies $d(v) \xrightarrow{*} \mathit{pxdom}(v)$.

An xdom path is also an sdom path and an sdom path is also a pxdom path. This implies the following relation:

$$\mathit{pxdom}(v) \xrightarrow{*} s(v) \xrightarrow{*} \mathit{xdom}(v)$$

(see Figure 3.3). If $\mathit{micro}(u) = \{u\}$, which is true if u is in C, then $\mathit{pxdom}(u) = \mathit{xdom}(u) = s(u)$. In order to locate the external dominators we only need to do computations in C.

Then the pushed external dominators account for parts of sdom paths inside a microtree. The next result suggests a way to computate immediate dominators from pushed external dominators.

Lemma 3.1 [BKRW98b] *For any v, there exists a $w \in micro(v)$ such that*

(1) $w \xrightarrow{*} v$;

(2) $pxdom(v) = pxdom(w)$;

(3) $pxdom(w) = s(w)$;

(4) $d(w) \notin micro(v)$.

Moreover, if $d(v) \notin micro(v)$ *then* $d(v) = d(w)$.

3.2.2 Overview of the algorithm

The BKRW algorithm consists of a preprocessing phase and two main phases of computation: the pxdom phase and the idom phase. The two main phases can be combined into a single phase, similarly to the LT algorithm. The preprocessing phase determines for each vertex $v \notin C$ whether $d(v) \in micro(v)$, and, if so, computes $d(v)$. The pxdom phase computes $pxdom(v)$ for each vertex v in D. Finally, the idom phase applies Lemma 3.1 to find $d(v)$ for each v such that $d(v) \notin micro(v)$.

The pxdoms are computed by processing the microtrees in reverse preorder. Each vertex $v \in D$ is associated with a value *value*(v), initially equal to v. After a microtree T is processed we have *value*$(v) =$ *pxdom*(v) for all $v \in T$. This is accomplished by executing procedure *pxdom_phase* of Figure 3.4.

When $b > a$ then all the vertices z that satisfy $\mathrm{NCA}(D, a, b) \xrightarrow{+} z \xrightarrow{*} b$ have been processed before T and the corresponding pxdoms are known. Procedure *push_values*(T) completes the computation of *pxdom*(u) for all $u \in$ *micro*(v) after all their xdoms are known. This is accomplished as follows. Let $G(T)$ be the graph induced by the vertices

```
$pxdom\_phase(G = (V, A, r), D)$

    for $v$ in reverse preorder of $D$ do
        $T \leftarrow micro(v)$
        $a \leftarrow p(root(T))$
        for $u \notin T$ such that $(u, v) \in A$ do
            $b \leftarrow \eta(u)$
            if $(b > a)$ then
                $x \leftarrow \min \left\{ value(z) \mid \mathrm{NCA}(D, a, b) \xrightarrow{+} z \xrightarrow{*} b \right\}$
            else $x \leftarrow \infty$ endif
            $value(v) \leftarrow \min \{value(u), value(v), x\}$
        done     /* at this point $value(v) = xdom(v)$ */
        if $v = root(T)$ then
            $push\_values(T)$
        endif     /* at this point $value(v) = pxdom(v)$ */
    done


$idom\_phase(D)$

    for $v$ in reverse preorder of $D$ do
        $a \leftarrow p(root(v))$
        if $(iidom(v) \notin micro(v))$ then
            $y \leftarrow \mathrm{argmin}_z(\left\{ value(z) \mid pxdom(v) \xrightarrow{+} z \xrightarrow{*} a \right\})$
            if $value(y) = value(v)$ then
                $idom[v] \leftarrow value(v)$     /* immediate dominator found */
            else
                $idom[v] \leftarrow y$     /* relative dominator found */
            endif
        endif
    done
```

Figure 3.4: Procedures that implement the two phases of the BKRW algorithm.

of microtree T and

$$Y_T(u) = \{w \in T \mid \text{there is a path from } w \text{ to } u \text{ in } G(T)\},$$

for any vertex u in T. Then we have

$$\mathit{pxdom}(u) = \min\{\mathit{xdom}(w) \mid w \in Y_T(u)\}.$$

So it suffices to find the strongly connected components of $G(T)$ and process them in topological order. Hence, *push_values* runs in linear time.

In order to compute the immediate dominators from the pxdom function we need to know whether $d(v) \in \mathit{micro}(v)$ for each v. We start by constructing from $G(T)$ an augmented graph $\mathit{aug}(T)$ as follows. We add a vertex t and for each arc (u, v) where $v \in T$ and $u \notin T$ we add the arc (t, v). Let $\mathit{iidom}(v)$ be the *internal immediate dominator* of a vertex $v \in T$, defined as the immediate dominator of v in $\mathit{aug}(T)$. We have

Lemma 3.2 [BKRW98b] *Let v be any vertex of a microtree T. Then $d(v)$ is in T if and only if $\mathit{iidom}(v) \neq t$.*

The *iidom*s can be computed by any simple (superlinear) dominators algorithm. Since there may be too many microtrees in D, BKRW cannot afford to run the simple algorithm for each microtree individually, but by using memoization it avoids repeating the same computations for identical graphs. In the worst case the algorithm will compute the *iidom*s for every possible $\mathit{aug}(T)$ of size $O(g^2)$ and use table lookups to retrieve the *iidom* values in constant time. If $g = O(\log^{1/3} n)$ this computation can be done in linear time. Although this process requires a RAM to implement memoization, Buchsbaum et al. showed how to achieve the same effect on a pointer machine, by introducing a data structure tool they call *pointer-based radix sort* [BKRW98a]; the idea is to sort the graph encodings of each $\mathit{aug}(T)$ using a variation of radix sort, implementable on a pointer machine. After the *iidom*s and *pxdom*s are available, the idom phase can compute the immediate dominators of every vertex v that satisfies $\mathit{iidom}(v) \notin micro(v)$. To that end, Lemma 3.1, implies that it suffices to find a vertex $y \in C[\mathit{pxdom}(v), a]$ with minimum $\mathit{pxdom}(y)$.

(This follows from statement (3) of Lemma 3.1 and Lemma 2.5.) If $\mathit{pxdom}(y) = \mathit{pxdom}(v)$ then $d(v) = \mathit{pxdom}(v)$. Otherwise y is a relative dominator of v, i.e., $d(v) = d(y)$. For vertices with relative dominators the calculation of their immediate dominators is completed in a preorder pass, as in done LT. Procedure *idom_phase* of Figure 3.4 summarizes the computations performed during the idom phase of the BKRW algorithm.

The computations of each x in *pxdom_phase* and of each y in the *idom_phase* involve evaluations of minima on paths of C. In order to carry out these computations the BKRW algorithm maintains (similarly to the Lengauer-Tarjan algorithm) a *link-eval* forrest on C [Tar79a] (see also Section 2.2).

The required operations on C are supported by a slightly modified version of Tarjan's *link-eval* data structure, which takes advantage of the following two facts: (a) the special order in which the *link* operations on C appear and (b) the sublinear number of leaves of C. However, in the following section we show that these facts do not suffice to get the desired linear time bound.

3.2.3 The problem in the analysis

In [BKRW98a, BKRW98b] it is observed that the link operations on C appear *bottom-up*, which means that v is linked to $p(v)$ only after all the vertices in the subtree rooted at v have been linked. Let T_1 and T_2 be the trees in F that contain $p(v)$ and v respectively. It is straightforward to verify that T_2 always contains at least one leaf in C and T_1 is either a singleton or also contains at least one leaf in C. The idea proposed in [BKRW98a, BKRW98b] is to make all the non-singleton trees in the virtual *link-eval* forest $\mathcal{F}$ (see Section 2.2) have roots which are leaves in C. By maintaining this invariant, *link* can be performed simply by linking two leaves or a single vertex to a leaf; similarly *eval* can be performed by evaluating minima on paths that are composed only of leaves of C. This property clearly implies $O(m\alpha(m, L) + n)$ running time. Now since $L < 2n/\log^{1/3} n$, we have $m/n = \Omega(\log^{1/3} n)$ and equation (1.1) together with the fact that for fixed n, $\alpha(m,n)$ decreases as m/n increases, imply that

$$O(m\alpha(m,L)+n) = O(m+n).$$

In fact, by (1.1), much smaller values of m/n would also suffice.

The problem with this analysis is that Tarjan's *link-eval* data structure [Tar79a] is not sufficiently flexible, in the sense that we are not free to choose the roots of the trees in $\mathcal{F}$ suitably so that the necessary invariant would hold. For example consider the case where D is just a path $(v_1 = r, v_2, \ldots, v_n)$. Then $C = (v_1, v_2, \ldots, v_k)$, where $k = n - g = O(n)$. Each successive *link* connects v_{i+1} to v_i, and v_i becomes the root of the tree that contains v_j, $i \leq j \leq k$. Suppose we have *pxdom*$(v_k) = v_1$. Then a careful look at the implementation of the *link-eval* structure in [Tar79a] reveals that v_k will only participate in the first *link* and all the other *link*s will involve internal vertices of C, while the analysis in [BKRW98b] assumes that every vertex will eventually be linked to v_k. In order to do so we need to update suitably the labels maintained by the *link-eval* structure after each *link* so that *eval* will return the correct result. However there is no way to carry out these updates efficiently; see Figure 3.5. Therefore, in this situation, we essentially apply the Lengauer-Tarjan algorithm on a graph with $O(n)$ vertices and $O(m)$ arcs, so the running time is not linear. We note that the special-case analysis in [BKRW98a, BKRW98b] for disjoint set union does apply to their off-line nearest common ancestors algorithm (since for this problem the *link-eval* structure does not need to maintain labels). In fact, we will use this result in our algorithm.

3.2.4 A fix for the random-access model

Buchsbaum et al. gave the following fix for their algorithm (Corrigendum of [BKRW98b]). The idea is to translate the *link-eval* operations on C to equivalent *link-eval* operations on C' and *path link-eval* operations inside each unary path. Since C' has $O(n/g)$ vertices, the corresponding *link-eval* operations take $O(n)$ time and in conjunction with the linear-time RAM solution of the path *link-eval* problem given in [AHLT99] they achieve an overall

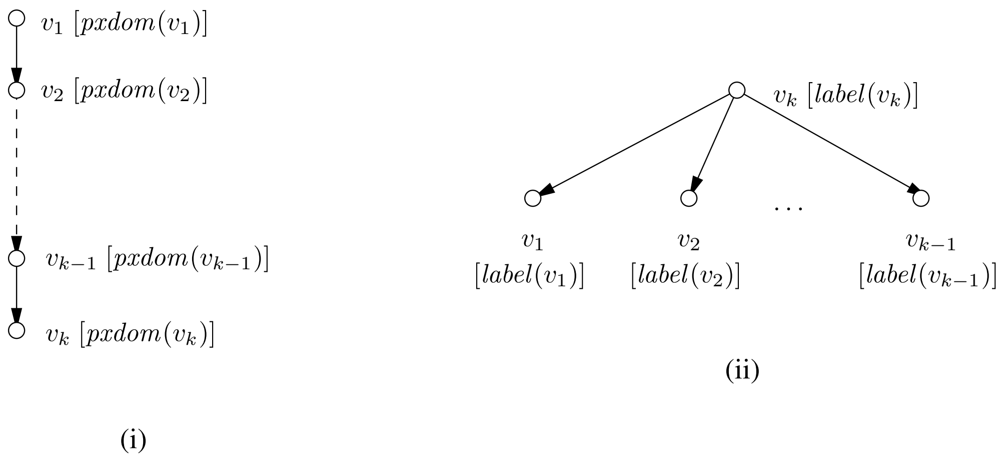


Figure 3.5: Maintaining the *link-eval* labels on a unary path. (i) C is a line $(v_1, \ldots, v_k)$. The value of *pxdom*(v_i) is known only after all descendants of v_i have been linked. (ii) The virtual forrest $\mathcal{F}$ that is maintained by the BKRW algorithm has only one non-singleton tree, which is rooted at v_k. After each v_i is linked to its parent, *label*(v_j) must be equal to a vertex with minimum *pxdom* on the path $C[v_i, v_j]$.

linear-time algorithm. Still this algorithm is much simpler than the Alstrup, Harel, Lauridsen and Thorup (AHLT) algorithm, since it avoids the need for complicated heap data structures.

The Alstrup et al. algorithm for path *link-eval* is based on a reduction of this problem to a special case of disjoint set union. This is accomplished as follows. Let $(v_1, \ldots, v_n)$ be the input path. Initially, before any *link*, each vertex comprises a distinct singleton set $\{v_i\}$. When v_i is linked to v_{i-1}, all the vertices v in $C[v_i, v_k]$ with *value*$(v) >$ *value*(v_i) are inserted into v_i's set. These vertices are found efficiently by maintaining a stack S containing the top vertex of each set that has been processed so far. The stack is initially empty, and v_i is pushed onto S after *link*(v_i) is executed. To perform *link*(v_i) we remove from the stack all the vertices v with value greater than *value*(v_i) and we execute *union*(v_i, v); also notice that *eval*(v_j) is simply a *find*(v_j) operation. Since the structure of the unions is known a priori, the linear-time DSU algorithm of Gabow and Tarjan [GT85] can be used to give a linear-time solution to the path *link-eval* problem. The Gabow and Tarjan DSU algorithm uses preprocessing of all possible small instances of the problem combined with table lookups, and therefore requires a RAM. Whether there is a linear-time pointer-machine solution for path *link-eval* is unresolved.

3.3 A linear-time pointer-machine algorithm

Our goal is to order the *pxdom* calculations of the BKRW algorithm appropriately, so that we can map the *eval*s on C to a combination of equivalent operations on C' and *off-line* NCA queries on fixed trees. Therefore, this modified algorithm essentially runs the Lengauer-Tarjan algorithm on C' and in conjunction with the linear-time pointer-machine preprocessing of small graphs of [BKRW98a] (via pointer-based radix sort) it achieves linear running time.

Algorithm PTRDOM of Figure 3.6 outlines our dominators algorithm. For clarity we present the algorithm as consisting of two main phases which (similarly to the LT and BKRW algorithms) can be combined into one phase. Also, PTRDOM runs the same preprocessing phase as BKRW. Again the first phase computes the *pxdom*s of each vertex in D. Essentially, we are processing the nodes of C' in reverse DFS order. For each node in C' we compute the *pxdom*s of the vertices that belong to the corresponding line ℓ and to the non-trivial microtrees adjacent to ℓ. The computations proceed in the following order:

(1) Compute *pxdom*(v) for all v that belong to non-trivial microtrees to the right of and below ℓ.

(2) Compute *pxdom*(v) for all v on ℓ. Then, carry out a preprocessing step that uses the *pxdom*s of the line to calculate some values that will be used off-line in step (3).

(3) Compute *pxdom*(v) for all v that belong to non-trivial microtrees to the left of ℓ.

Notice that for each vertex we compute the same information as in the BKRW algorithm (i.e., its *pxdom*), but the order of the computations is different. Specifically, when the BKRW algorithm processes a vertex v of the core, it has already computed the *pxdom*s of all vertices $u > v$. In our new algorithm, however, *pxdom*(u) is known only if u is not in *TL*(*line*(v)).

We process each non-trivial microtree T as done in the BKRW algorithm; procedure *process_microtree*, shown in Figure 3.7, implements the necessary computations. Procedure *process_line* computes the semi-dominators of a line ℓ before processing *TL*(ℓ). This enables us to pre-compute minima on paths that lie entirely inside ℓ (this constitutes the second part of step(2)) and retrieve these results in constant time when we process the microtrees in *TL*(ℓ) (step (3)). The second phase uses the *pxdom*s to compute the immediate dominators. For each $v \in D$, PTRDOM either computes $d(v)$ or determines a proper ancestor u of v such that $d(v) = d(u)$. Again we evaluate minima on paths of C using the same methods we applied in phase one.

It is crucial to note that we use the *pxdom_phase* procedure of the BKRW algorithm only to compute *pxdom*s of vertices in non-trivial microtrees. The remainder of this section details the computation of *pxdom*s (and hence semidominators) on vertices in C.

3.3.1 Evaluating minima on paths of the core

We describe two link-eval data structures: one that operates on C and one that operates on C'. To distinguish the two, we use *vlink* and *veval* (for *virtual link* and *eval*) to operate on C and *link* and *eval* to operate on C'. Algorithm `IDOM` and its various subroutines work on C and hence call *vlink* and *veval*, which in turn work on C' and hence call *link* and *eval*. The semantics of the respective operations are described in Section 2.2; we will define the values of nodes in the forests shortly. That is, the abstract data structures are identical, except for their respective applications to C and C'. We use different concrete implementations, however, to achieve linear running time.

We use Tarjan's [Tar79a] *link-eval* data structure to implement *link* and *eval* operations on C'. We associate with each vertex $v \in D$ a value *value*(v). Initially we assign *value*$(v) \leftarrow v$, and after processing v (during the first phase) we will have *value*$(v) =$ *pxdom*(v). Remember that the vertices of C' correspond to lines in C. We define the value

```
**Algorithm** PTRDOM($G = (V, A, r)$)
    /* Preprocessing */
    *partition*($G$)
    **for all** $v$, compute *iidom*($v$)

    /* Phase 1 */
    **for** $v \in V - r$ in reverse preorder of $D$ **do**
        **if** *micro*($v$) is non-trivial **then**
            **if** $v$ is the largest vertex in *micro*($v$) **then**
                *process_microtree*($v$)
            **endif**
        **else**
            **if** $v$ = *bottom*(*line*($v$)) **then**
                *process_line*($v$)
                prepare off-line values for *veval* in *line*($v$)
            **endif**
            *vlink*($v$)
        **endif**
        *process_vertex*($v$)
    **done**

    /* Phase 2 */
    re-initialize the *vlink* forest
    **for** $u \in C$ in reverse preorder **do**
        **if** $u$ = *bottom*(*line*($u$)) **then**
            prepare off-line values for *veval* in *line*($u$)
        **endif**
        *process_bucket*(*bucket*[$u$])
        *vlink*($u$)
    **done**
```

Figure 3.6: Algorithm PTRDOM.

```
$\textit{process\_microtree}(v)$

    for $w \in \textit{micro}(v)$ do
        for $u \notin \textit{micro}(v)$ such that $(u, w) \in A$
            if $\eta(u) \leq \eta(v)$ then
                If $u \in C$ then $x \leftarrow u$ else $x \leftarrow \textit{value}(u)$ endif
            else
                $x \leftarrow \min\{\textit{value}(u), \textit{value}(\textit{veval}(\eta(u)))\}$
            endif
            $\textit{value}(w) \leftarrow \min\{\textit{value}(w), x\}$
        done
    done
    $\textit{push\_values}(\textit{micro}(v))$


$\textit{process\_vertex}(v)$

    if $\textit{iidom}(v) \in \textit{micro}(v)$ then
        $\textit{idom}[v] \leftarrow \textit{iidom}(v)$
    else
        add $v$ to $\textit{bucket}[\textit{pxdom}(v)]$
    endif


$\textit{process\_bucket}(\textit{bucket}[u])$

    for $v \in \textit{bucket}[u]$ do
        if $u = p_D(\textit{root}(\textit{micro}(v)))$ then
            $z \leftarrow v$
        else
            $z \leftarrow \textit{veval}(p_D(\textit{root}(\textit{micro}(v))))$
        endif
        if $\textit{pxdom}(z) = u$ then $\textit{idom}[v] \leftarrow u$ else $\textit{idom}[v] = \textit{idom}[z]$ endif
    done
```

Figure 3.7: Subroutines for processing non-trivial microtrees, vertices and buckets.

of each vertex $\ell \in C'$ for the *link-eval* forest as follows. Consider a line ℓ, and let v be the last vertex in ℓ that was linked. Then all the descendants of v in ℓ are also linked. We define the (current) label of a line ℓ, denoted by *linelabel*(ℓ)[1] , to be a vertex in the tree path from v to *bottom*(ℓ) with minimum pxdom; that is,

$$\mathit{linelabel}(\ell) = \operatorname*{argmin}_{u \in \ell,\ u \geq v} \mathit{pxdom}(u).$$

If no vertex in ℓ has been linked then we assign *linelabel*$(\ell) \leftarrow \infty$ and assume that *value*$(\infty) = \infty$. Then the value that is used for each $\ell \in C'$ by the *link-eval* structure is *value*(*linelabel*(ℓ)). After computing the *pxdom*s for all the vertices in ℓ we need to store for each $v \in \ell$ a value *up*(v), which is a vertex in the tree path from *top*(ℓ) to v with minimum *pxdom*, i.e.,

$$\mathit{up}(v) = \operatorname*{argmin}_{u \in \mathit{line}(v),\ u \leq v} \mathit{pxdom}(u).$$

Initially we assign *up*$(v) \leftarrow \infty$ for all $v \in \ell$.

The implementation of *vlink* and *veval* is shown in Figure 3.8. Let F be the forest that is built by *vlink* on the vertices of C. We use *value*(v) as the value of each vertex v in F. For any vertex v in F, $r_F(v)$ denotes the root of the tree in F that contains v.

Note that *vlink* and *veval* effect the abstract *link-eval* operations on C. Our algorithm does not need to maintain explicitly the *vlink* forest F, because the vertices in C are linked in reverse DFS order and it suffices to keep track only of the last vertex in C for which *vlink* was performed. To that end the *vlink-veval* data structure maintains an internal variable *lastlinked*, i.e., this variable is accessed only by *vlink* and *veval*. Initially, before any *vlink*, we assign *lastlinked* $\leftarrow \infty$.

We observe that *veval* can calculate minima on paths in C that extend beyond a single line by using the line labels and the *up* values. Consider a call to *veval*(v). Notice that if p_F(*lastlinked*) $\neq r_F(v)$ then $r_F(v)$ must be the bottom vertex of a line; since we operate on the core tree, only the bottom vertex of each line ℓ can have children outside ℓ. See

[1]We use this notation for the label of a line to avoid confusion with the labels used inside the *link-eval* structure of Section 2.2.

```
$vlink(v)$

    $\ell \leftarrow line(v)$
    **if** $value(linelabel(\ell)) > value(v)$ **then** $linelabel(\ell) \leftarrow v$ **endif**
    **if** $v = top(\ell)$ **then** $link(\ell)$ **endif**
    $lastlinked \leftarrow v$


$veval(v)$

    **if** $v < lastlinked$ **then return** $v$ **endif**
    $\ell \leftarrow line(v)$
    $q \leftarrow line(lastlinked)$
    **if** $q = \ell$ **then**
        **if** $lastlinked = top(\ell)$ **then return** $up(v)$
        **else return** $\mathrm{RMQ}(lastlinked, v)$ **endif**
    **else**
        $x \leftarrow linelabel(eval(p_{C'}(\ell)))$
        **if** $value(x) > value(up(v))$ **then** $x \leftarrow up(v)$ **endif**
        **if** $(lastlinked \xrightarrow{+} v)$ **and** $(value(linelabel(q)) < value(x))$ **then**
            $x \leftarrow linelabel(q)$
        **endif**
        **return** $x$
    **endif**
```

Figure 3.8: Operations on C.

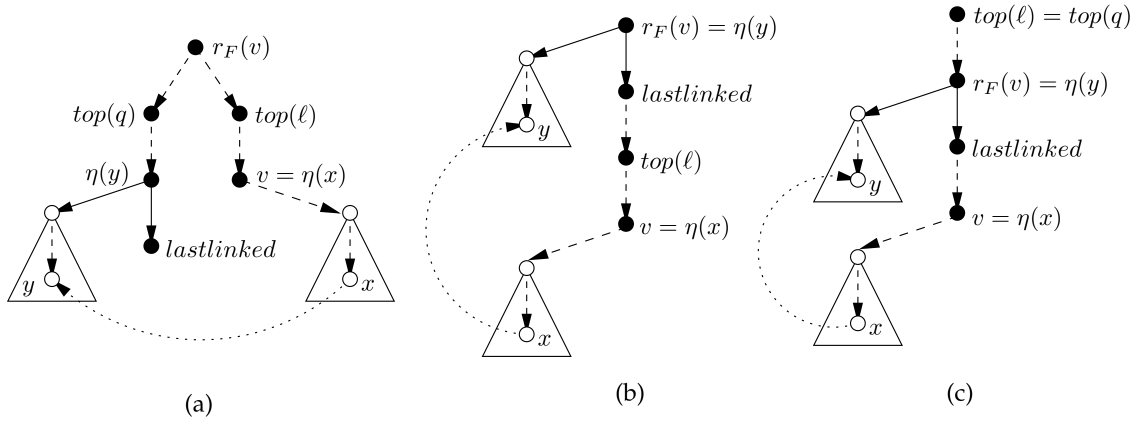


Figure 3.9: Examples of how the *vlink-veval* structure evaluates minima on paths of C. Dashed arcs represent (possibly nil) tree paths; solid arcs are tree arcs; dotted arcs are non-tree arcs; filled nodes are in the core; open nodes are in non-trivial microtrees; triangles denote non-trivial microtrees. In all the examples y is the currently processed vertex and belongs to a non-trivial microtree; $\ell = \mathit{line}(v)$ and $q = \mathit{line}(\mathit{lastlinked})$. The arc (x, y) causes a call to $\mathit{veval}(v)$, where $v = \eta(x)$.

Figure 3.9(a) for an example. In this case it suffices to consider the values of $\mathit{up}(v)$ and $\mathit{eval}(p_{C'}(\ell))$. Otherwise, when $p_F(\mathit{lastlinked}) = r_F(v)$, we may have $top(q) \xrightarrow{+} \mathit{lastlinked}$, in which case q is not linked (e.g., Figure 3.9(b)). Then we also need to consider the values in $\mathit{lastlinked} \xrightarrow{*} \mathit{bottom}(q)$, which is done in the last **if** statement in the code for *veval*.

Obviously we cannot apply similar computations to calculate path minima inside a line. In this case we take advantage of the following observation. Suppose that we need to compute a vertex with minimum *pxdom* in a tree path inside ℓ from $v_i \neq top(\ell)$ to $v_j \neq \mathit{bottom}(\ell)$. Then, this computation is performed in one of the following cases:

(i) During the pxdom phase, when we compute $\mathit{pxdom}(v_{i-1})$.

(ii) During the pxdom phase, when we compute $\mathit{pxdom}(y)$ for some vertex $y \in TL(v_i)$ (e.g., Figure 3.9(c)).

(iii) During the idom phase, when we compute $d(x)$ where x is such that $\eta(x) = v_j$ and $\mathit{pxdom}(x) = v_{i-1}$.

(The fact that during the pxdom phase only cases (i) and (ii) may apply is implied by the properties of DFS.) In the next section we show that by processing ℓ appropriately we can compute $\mathit{pxdom}(v)$ for all $v \in \ell$ without knowing $\mathit{pxdom}(u)$ for any $u \in \mathit{TL}(\ell)$, so this takes care of case (i). Then the task of computing path minima inside $\ell = (v_1, \ldots, v_k)$ for cases (ii) and (iii) is equivalent to the *Range Minimum Query* problem (RMQ). In an instance of this problem we are given an array $A = \begin{bmatrix} a_1 & a_2 & \ldots & a_n \end{bmatrix}$ and a set Q_A of queries. A query is a pair of array indices (i, j), such that $1 \leq i \leq j \leq n$. For each pair $(i, j) \in Q_A$ we want to find the index of the smallest element in the subarray $A(i : j) = \begin{bmatrix} a_i & a_{i+1} & \ldots & a_j \end{bmatrix}$. We denote by $\mathrm{RMQ}(A, i, j)$ the answer to the query $(i, j) \in Q_A$. In our case $A = \begin{bmatrix} \mathit{pxdom}(v_1) & \ldots & \mathit{pxdom}(v_k) \end{bmatrix}$, and we denote a query for the minimum *pxdom* in the path $C[v_i, v_j]$ by $\mathrm{RMQ}(v_i, v_j) \equiv \mathrm{RMQ}(A, i, j)$. Note that in both (ii) and (iii) we have $\mathit{lastlinked} = v_i$, so $\mathrm{RMQ}(\mathit{lastlinked}, v_j)$ is the desired value of the corresponding computation. Since we have $\mathit{line}(v_j) = \mathit{line}(\mathit{lastlinked})$ and $\mathit{lastlinked} \neq \mathit{top}(\mathit{line}(v_j))$, $\mathit{veval}(v_j)$ returns the correct value.

We can solve an instance of the RMQ problem by computing NCAs in the Cartesian tree T_ℓ [Vui80] for the sequence of values $\mathit{pxdom}(v_i)$, $1 \leq i \leq k$. The Cartesian tree for the array $A = \begin{bmatrix} a_1 & \ldots & a_k \end{bmatrix}$ is a binary tree defined recursively as a tree rooted at $j = \mathrm{RMQ}(A, 1, k)$ with its left subtree being a Cartesian tree for $A(1 : j)$ and right subtree being a Cartesian tree for $A(j+1 : k)$. In [GBT84] it is shown how to construct a Cartesian tree incrementally in $O(k)$ time; the Cartesian tree for $A(1 : i)$ is constructed from the tree for $A(1 : i-1)$ by ascending the rightmost path until a node j such that $a_j < a_i$ is reached. If there no such node then i is made root of $A(1 : i)$ with left subtree the Cartesian tree for $A(1 : i - 1)$. Otherwise we make the right subtree of j the left subtree of i, and i is made the right child of j. Note that this construction is implementable on a pointer machine.

There are RAM solutions to the off-line NCAs problem on a tree T that answer each query in constant time using $O(|T|)$ time for preprocessing [HT84, SV88, BFC00]. On a pointer machine we can use the algorithm of Buchsbaum et al. [BKRW98a], that answers all the NCA queries of a query set Q_T in $O(|T| + |Q_T|)$ time. We note that this linear-time pointer-machine algorithm requires Q_T to be known a priori, while the RAM algorithms do not have this restriction. Fortunately, in our case that query sets are indeed known off-line. Each range minimum query that needs to be answered during the first phase of Algorithm PTRDOM corresponds to an arc $(u, v) \in A$, where $u > v$, *line*$(\eta(u)) =$ *line*$(\eta(v))$, and neither $\eta(u)$ nor $\eta(u)$ is an endpoint of a line. The values that are computed by the NCA algorithm are associated with the corresponding arcs and can be retrieved in constant time when needed. In the second phase we make an NCA query for every vertex v such that both *pxdom*(v) and $\eta(v)$ are in the same line and neither is an endpoint. So again the query set for each line is given before we begin the NCA calculations. The values that are computed by the NCA algorithm are associated with the corresponding vertices that are processed and can be retrieved in constant time.

To clarify: The results of the RMQ operations are computed before the corresponding *veval*s are actually executed. To that end, the first phase of Algorithm PTRDOM delays processing each left microtree *TL*(ℓ) until line ℓ is processed, since then all the induced NCA operations can be performed off-line. Similarly, in phase two, PTRDOM computes the results of the RMQs corresponding to ℓ before any bucket associated with a vertex in ℓ gets processed.

3.3.2 Computing semidominators in a line

Consider a line $\ell = (s = v_1, \ldots, v_k = t)$. Here we describe a procedure *process_line* that computes the semidominators of the vertices in ℓ without evaluating *pxdom*(u) for any u in *TL*(ℓ). For more clarity, first we give an algorithm that consists of two phases and prove its correctness. Then we show how to order the calculations in a single phase, thus lowering the constant factors of our algorithm.

The first phase of *process_line* transforms the input graph G to a graph $G_\ell = (V, A_\ell, r)$ that has the following two properties:

(i) None of the vertices in ℓ has a predecessor in *TL*(ℓ).

(ii) Each v in ℓ has the same semidominator in G_ℓ as in G.

The second phase runs a modified version of the pxdom phase of BKRW for ℓ on G_ℓ. Since in the modified graph there is no arc entering ℓ from $TL(\ell)$, the algorithm will not need the *pxdom* values of these vertices. Also, the fact that the set of vertices in ℓ are linearly ordered allows us to apply a simple and efficient alternative to the *eval* function for the evaluation of path minima inside ℓ.

First phase

We process the line bottom-up, and for each vertex v in ℓ we only consider those arcs (u, v) that enter v from $TL(\ell)$. In order to keep track of the visited vertices in $TL(\ell)$ we use a mark bit; initially all vertices are unmarked.

Suppose that we are currently processing v_i and let $u \in TL(v_j)$ be a predecessor of v_i. Then, $i \leq j < k$ and $\eta(u) = v_j$. Let $w = root(micro(u))$. We simulate a collapse of the tree path $D[w, u]$ as follows. We start from u and walk backwards until we reach w or a marked vertex. For each vertex x that we visit in this walk, we mark x and for each arc (y, x) we insert (y, v_i) at the end of the list of incoming arcs of v_i. Finally, we add the arc (v_j, v_i). The added arcs originating from $TL(\ell)$ will eventually be examined in this phase while processing v_i. Let A^+ be the list of the new arcs, ordered from the first arc that was added during this phase to the last.

We denote by $G^{(i)}$ the modified graph G after processing $v_k, v_{k-1}, \ldots, v_i$. We use similar notation for other quantities. Note that $G^{(k)} = G$. The graph G_ℓ is formed from $G^{(1)}$ by removing all the arcs entering ℓ from $TL(\ell)$. Then, property (i) trivially holds. Now we need to show that G_ℓ also satisfies property (ii).

For any vertex v in ℓ, let $scc(v)$ be the strongly connected component of G that contains only vertices that are descendants of v in the DFS tree D. Let u be a vertex in $TL(\ell)$. We denote by $scc^{-1}(u)$ the maximum vertex in ℓ such that $u \in scc(v)$. If there is no such v for u we leave $scc^{-1}(u)$ undefined. Note that u is marked if and only if $scc^{-1}(u)$ is defined. Furthermore, u is marked while $v_i = scc^{-1}(u)$ is processed and therefore $G^{(j)}$ contains an arc (x, v_i) for each predecessor x of u, where j is any integer in $[1, i]$.

The next lemma shows that the semidominators remain the same in each $G^{(i)}$.

Lemma 3.3 *For any v in ℓ and any integer i in $[1, k]$, $s^{(i)}(v) = s(v)$.*

Proof: We prove the Lemma by induction on the rank of the added arcs in A^+; for each rank $i \in [1, |A^+|]$, we want to show that the addition of the first i arcs in A^+ does not change the semidominator of any vertex.

Let G' be the graph after adding the first $i-1$ arcs of A^+. Let (u, v) be the i-th arc. Then G' must contain the arcs (u, x) and (y, v), where x and y are vertices in $TL(\ell)$ that satisfy $v \xrightarrow{+} x \xrightarrow{*} y$ and $v = scc^{-1}(x) = scc^{-1}(y)$. Now consider any sdom path P in $G' + (u, v)$ that starts from a vertex w and enters v using the new arc (u, v). Then, the prefix of P up to u followed by (u, x), $D[x, y]$ and (y, v) is an sdom path from w to v that also exists in G'. Hence, the semidominator of v does not change after the addition of (u, v). The previous argument works both for the basis of the induction (where $i = 0$ and $G' = G$) and for the induction step. $\square$

We will use the previous fact to show that for any v in ℓ there is an sdom path from $s(v)$ to v in G_ℓ. To that end we first need two results about the strong components of the intermediate graphs $G^{(i)}$ and the final graph G_ℓ. The first result shows that no $scc(v)$ obtains any new vertex as the arcs in A^+ are added to the graph.

Lemma 3.4 *For any $v \in \ell$ and any integer j in $[1, k]$, $scc^{(j)}(v) = scc(v)$.*

Proof: Let *cycle*(v) be the set of vertices z such that (z, v) is a cycle arc (w.r.t. the DFS tree D) entering v. Then $scc(v)$ is the set of vertices w that are descendants of v in D and such that there is a path from w to some $z \in$ *cycle*(v). Hence, we only need to consider the new cycle arcs that are added during the first phase. Our goal is to show that the vertices that are marked while processing a cycle arc (u, v) are in $scc(v)$. This statement implies the Lemma, since then any predecessor of such a marked vertex that is a descendant of v must also belong to $scc(v)$.

We will use induction on the order the cycle arcs are processed. For each such arc (u, v) we claim that u already belonged to $scc(v)$ in G. Given this, the fact that the vertices

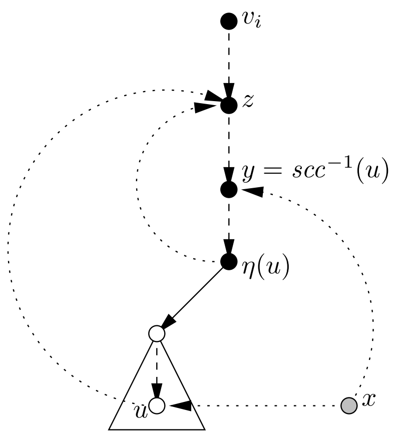


Figure 3.10: Proof of Lemma 3.5. Dashed arcs represent (possibly nil) tree paths; solid arcs are tree arcs; dotted arcs are non-tree arcs; filled nodes are in the core; open nodes are in non-trivial microtrees; gray nodes can be in either state; triangles denote non-trivial microtrees where they are determined. Starting from a path containing only vertices in $scc(v_i)$ that connects a vertex $w \in scc(v_i)$ to v_i and goes through $u \in TL(\ell)$, we can find another path inside $scc(v_i)$ from w to v_i that avoids u.

that are marked while processing (u, v) are in $scc(v)$ follows immediately. Therefore, the claim implies that all the vertices that are marked while processing v must belong to $scc(v)$. Note that the claim is obvious for the first cycle arc that was processed, since this arc must exist in G. Hence, the base case holds. For the induction step it suffices to consider a cycle arc $(u, v) \notin A$, since otherwise the same argument with the base case applies. The fact that (u, v) was not present in G implies that u is a predecessor of an already marked vertex w; this vertex must have been marked while v was processed, hence $w \in scc(v)$ by induction hypothesis. Since u is a descendant of v, we conclude that $u \in scc(v)$ as claimed. □

Lemma 3.5 *Let v_i and v_j be any vertices in $\ell = (v_1, \ldots, v_k)$ such that $v_j \in scc(v_i)$. Then, there is a path in $G^{(i)}$ from v_j to v_i that avoids $TL(\ell)$ and contains only vertices in $scc(v_i)$.*

Proof: We prove by induction on i that for any $v_j \in \ell \cap scc(v_i)$, there is a path in $G^{(i)}$ from v_j to v_i that avoids $TL(\ell)$ and contains only vertices in $scc(v_i)$.

The base case $i = k$ is immediate since v_k is the only vertex of $\ell \cap scc(v_k)$.

For the induction step we note that since $v_j \in \ell \cap scc(v_i)$ there is a path from v_j to v_i in $G^{(i+1)}$ (and in all the intermediate graphs $G^{(k)}, \ldots, G^{(i+2)}$) that contains only descendants of v_i in D. Let P_μ be such a path with minimum number of vertices in $TL(\ell)$, where μ is the number of vertices in $P_\mu \cap TL(\ell)$.

If $\mu = 0$ then the result is immediate. Otherwise, let u be the last vertex on P_μ that is in $TL(\ell)$. Let z be the successor of u on P_μ. Then, by the definition of P_μ and u (and by the properties of DFS; Lemma A.1) we have that $v_i \xrightarrow{*} z \xrightarrow{*} \eta(u)$. Also, let x be the predecessor of u on P_μ and let $y = scc^{-1}(u)$ (see Figure 3.10). Then, $v_i \xrightarrow{*} z \xrightarrow{*} y \xrightarrow{*} \eta(u)$. If $v_i \neq y$ then by construction $G^{(i+1)}$ contains the arcs (x, y) and $(\eta(u), z)$. Consider the path Q in $G^{(i+1)}$ formed by the prefix of P_μ up to x, followed by (x, y), $y \xrightarrow{*} \eta(u)$, $(\eta(u), z)$, and the suffix of P_μ from z to v_i. This path contains only descendants of v_i in D and avoids u. Thus, Q has has at most $\mu - 1$ vertices in $TL(\ell)$, a contradiction. Therefore, u is marked while processing v_i, which implies $v_i = z = y$ and $G^{(i)}$ contains the arc (x, v_i). Now consider the path $P_{\mu'}$ in $G^{(i)}$ formed by the prefix of P_μ up to x and followed by (x, v_i). This path connects v_j to v_i visiting only descendants of v_i in D, and also avoids u. So, $P_{\mu'}$ contains at most $\mu' \leq \mu - 1$ vertices in $TL(\ell)$. Then, we can apply the same analysis as for P_μ. By repeating these arguments it is clear that $G^{(i)}$ contains a path P_0 that satisfies the Lemma. □

The next lemma immediately implies that G_ℓ satisfies property (ii).

Lemma 3.6 *For any v_i in ℓ and any integer j in $[1, i]$, there is an sdom path in $G^{(j)}$ from $s(v_i)$ to v_i that does not pass though $TL(\ell)$.*

Proof: Clearly it suffices to show that such an sdom path for $v = v_i$ exists in $G^{(i)}$, since no path disappears during the processing of the ancestors of v_i in ℓ.

Consider an sdom path in $G^{(i)}$ from $s(v)$ to v and let u be the first vertex on P that is in $TL(\ell)$. If no such u exists then the Lemma clearly holds. Otherwise, let $y = scc^{-1}(u)$ and let x be the predecessor of u on P. Since u was marked when y was processed, $G^{(i)}$ contains the arc (x, y). Moreover, $y \in scc(v)$ so by Lemma 3.5 there is a path Q from y to

v that avoids $TL(\ell)$ and contains only descendants of v. Hence, the prefix of P until x, followed by (x, y) and Q is an sdom path from $s(v)$ to v that avoids $TL(\ell)$. □

Second phase

Suppose that the second phase of *process_line* runs the pxdom part of the BKRW algorithm for ℓ on G_ℓ. Since $pxdom(v)$ for any $v > v_k$ is already computed before the call to *process_line*, it follows from Lemma 3.6 and the correctness of the BKRW algorithm, that this algorithm correctly computes the semidominators for the vertices of ℓ. Recall that the BKRW algorithm uses a version of the $eval$ function that operates on C. So, in order to achieve linear running time for our overall dominators algorithm, we have to ensure that the $eval$ function operates only outside ℓ.

To that end, we employ a simple stack-based mechanism, similar to Gabow's linear-time algorithm for strong components [Gab00], in order to compute minimum semidominator values for paths that lie on the line. We make the observation that each evaluation of semidominator minima on a path $C[v_{i-1}, v_j]$ of ℓ is caused by a back arc (x, v_i), where x is vertex whose nearest ancestor in ℓ is v_j. This arc defines a strongly connected component in which v_i is the minimum vertex. We use a stack S to keep track of these strongly connected components. Initially the stack is empty, and a vertex v_i is pushed onto the stack after we are finished processing it. Each element of the stack corresponds to a strongly connected component *scc* of successive vertices $v_i, \ldots, v_j$ in ℓ, and can be represented by the vertex v_i which has the minimum DFS number in the component. Since $v_i = \min\{u \mid u \in \mathit{scc}\}$ we also have $s(v_i) = \min\{s(u) \mid u \in \mathit{scc}\}$. Suppose that when we reach vertex $v \in \ell$ the current status of the stack is $S = (u_d, u_{d-1}, \ldots, u_1)$, where $v = p_C(u_d)$ and $u_i < u_{i-1}$ for $2 \leq i \leq d$. Also, let $u \in \ell$ be a descendant of v. In order to evaluate the minimum semidominator value in the path $P = C[v, u]$ we simulate the contraction of P by performing a sequence of *pop* operations on S, until we reach a proper descendant of u. As we pop each element $w \leq u$ of S, we also perform the update $\mathit{value}(v) = \min\{\mathit{value}(v), \mathit{value}(w)\}$.

We process the arc (u, v_i) as follows. First we compute $x \leftarrow \mathit{veval}(u)$. Then we apply the stack-based contraction of $C[v_i, v_j]$. Finally, we set

$$\mathit{value}(v_i) \leftarrow \min\{\mathit{value}(x), \mathit{value}(v_i), \mathit{value}(u)\}.$$

Now we prove the correctness of our algorithm.

Lemma 3.7 *After process_line completes processing the line $\ell = (s = v_1, \dots, v_k = t)$ we have value$(v_i) = s(v_i)$, for $1 \leq i \leq k$, assuming that value$(u) =$ pxdom(u) for $u > t$.*

Proof: We will show that by processing the arcs with the stack-based contractions and the *veval* function produces the same result as the *eval* of BKRW. Then, as we have already mentioned, the correctness of our algorithm will follow from Lemma 3.6 and the correctness of the BKRW algorithm [BKRW98b, Theorem 4.4]. Recall that *lastlinked* is the last vertex in C on which *vlink* was applied. In particular this means *lastlinked* $=$ *top*(*line*(*lastlinked*)) and *lastlinked* $> t$.

Let (u, v_i) be the arc that is currently processed by *process_line*. Suppose that $\eta(u)$ is either an ancestor of v_i or unrelated to v_i. Since no vertex is linked during the execution of *process_line*, each call to *veval* returns the same value as the equivalent call to *eval* of BKRW.

Now suppose $\eta(u)$ is a descendant of v_i. The BKRW algorithm would handle this case by finding a vertex $z \in D(v_i, \eta(u)]$ with minimum semidominator. If $t \xrightarrow{+} \eta(u)$ then *veval*$(\eta(u))$ returns a vertex z with minimum value such that $t \xrightarrow{+} z \xrightarrow{*} \eta(u)$. Otherwise, *veval*$(\eta(u))$ returns $\eta(u)$. It remains to evaluate the path $D(v_i, v_j]$. This computation is performed by the stack mechanism. Let $v_{j'}$ be the new top vertex of the stack, after removing the vertices less or equal to v_j. Let w be the parent of $v_{j'}$ in ℓ. We have $w \geq v_j$, so the stack process has actually returned the minimum semidominator in $v_i \xrightarrow{*} w$. However, all vertices in $D[v_j, w]$ belong to *scc*(v_i). Hence $s(v_i) \leq s(z)$, for any $z \in D[v_j, w]$, and the stack-based computation returns the correct value. □

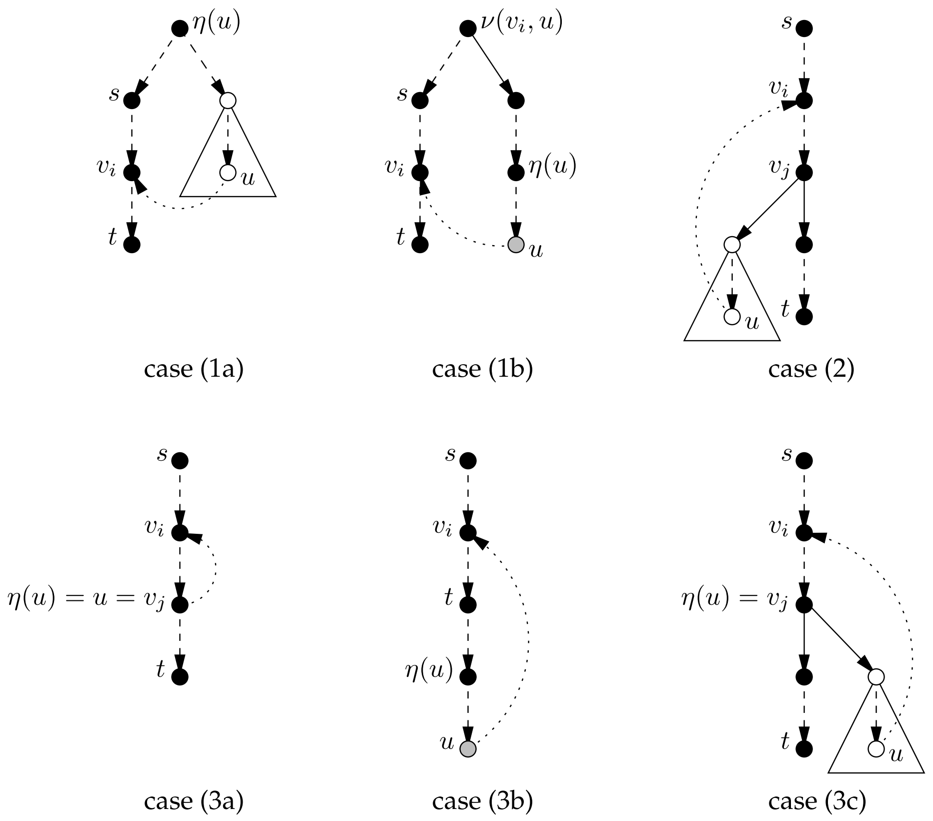


Figure 3.11: Computing the semidominators of the line $\ell = (s = v_1, \dots, v_k = t)$. Dashed arcs represent (possibly nil) tree paths; solid arcs are tree arcs; dotted arcs are non-tree arcs; filled nodes are in the core; open nodes are in non-trivial microtrees; gray nodes can be in either state; triangles denote non-trivial microtrees where they are determined. In case (1) either u and v_i are unrelated, or (u, v_i) is a forward arc: (1a) $\eta(u) \xrightarrow{+} v_i \Rightarrow$ *veval*$(\eta(u)) = \eta(u)$; (1b) $\eta(u)$ and v_i are unrelated $\Rightarrow$ *veval*$(\eta(u))$ returns a vertex $z \in D(\nu(v_i, u), \eta(u)]$ with minimum value (where $\nu(v_i, u) = \mathrm{NCA}(D, \{v_i, u\})$). In case (2) $u \in TL(v_j)$ where $v_i \xrightarrow{*} v_j \xrightarrow{+} t$. In case (3) $v_i \xrightarrow{+} u$ and $u \notin TL(\ell)$: (3a) $u = v_j \Rightarrow$ *veval*$(\eta(u)) = \eta(u)$; (3b) $t \xrightarrow{+} u \Rightarrow$ *veval*$(\eta(u))$ returns a vertex $z \in D(t, \eta(u)]$ with minimum value; (3c) $u \in TR(v_j) \Rightarrow$ *veval*$(\eta(u)) = \eta(u)$.

One-phase implementation of *process_line*

In this section we describe how to order the calculations of *process_line* in a single phase. Again, let $\ell = (s = v_1, \ldots, v_k = t)$ be the line currently processed. We ascend the path from t to s, and at each vertex v_i we examine the arcs entering v_i. Let u be the currently examined predecessor of v_i. Depending on the location of u we have the following cases. (See Figure 3.11.)

(1) u is unrelated to v_i, or an ancestor of v_i. We compute $x \leftarrow \mathit{veval}(\eta(u))$. Then we update

$$\mathit{value}(v_i) \leftarrow \min\{\mathit{value}(v_i), \mathit{value}(u), \mathit{value}(x)\}.$$

(2) $u \in \mathit{TL}(v_j)$, where $i \leq j < k$. We handle this case by simulating a contraction of the tree path $D[v_i, u] = C[v_i, v_j] \cdot D[v_j, u]$ to a single vertex v_i. For the $C[v_i, v_j]$ part we set $\mathit{value}(v_i)$ to be the minimum value in this path; the effect of the contraction is achieved by removing from S the proper descendants of v_i in ℓ until v_j. For the $D[v_j, u]$ part we start from u and walk backwards until we reach v_j or a marked vertex. For each vertex $w \neq v_j$ that we visit in this walk, we mark w and for each arc (z, w) we insert (z, v_i) at the end of the list of incoming arcs of v_i.

(3) $v_i \xrightarrow{+} u$ and $u \notin \mathit{TL}(\ell)$. We compute $x \leftarrow \mathit{veval}(\eta(u))$. Then we simulate a contraction of the tree path $C[v_i, \min\{t, \eta(u)\}]$ as in case (2), and set $\mathit{value}(v_i)$ to be the minimum value in this path. Finally we update

$$\mathit{value}(v_i) \leftarrow \min\{\mathit{value}(x), \mathit{value}(v_i), \mathit{value}(u)\}.$$

Figures 3.12 and 3.13 show an implementation of the single-phase version of ProcessLine. The correctness of this implementation is implied by Lemma 3.7 and the fact that Lemmas 3.3 and 3.6 hold for the intermediate graphs $G^{(i)}$.

```
process_line(t)
    $S \leftarrow \emptyset$
    $s \leftarrow top(line(t))$
    $v \leftarrow t$
    while $v \geq s$ do
        for $u \in pred(v)$ do
            if $v$ and $\eta(u)$ are unrelated then
                $x \leftarrow veval(\eta(u))$
                $value(v) \leftarrow \min\{value(v), value(u), value(x)\}$
            elseif $\eta(u) \in line(t)$ and $u \in TL(\eta(u))$ then
                $contract(v, \eta(u))$
                $backwalk(u, v)$
            else
                $x \leftarrow veval(\eta(u))$
                $contract(v, \min\{t, \eta(u)\})$
                $value(v) \leftarrow \min\{value(v), value(u), value(x)\}$
            endif
        done
        $push(S, v)$
        $v \leftarrow p_C(v)$
    done
    compute in a top-down pass $up(v)$ for all $v \in line(t)$
```

Figure 3.12: Procedure *process_line* computes the semidominators of a line ℓ given the *pxdom*s of the vertices $v > bottom(\ell)$.

```
$contract(u, v)$
    $w \leftarrow top(S)$
    while $w \leq u$ do
        $pop(S)$
        $value(v) \leftarrow min\,\{value(v), value(w)\}$
        $w \leftarrow top(S)$
    done
```

```
$backwalk(u, v)$
    while $mark(u) = false$ and $u \notin C$ do
        $mark(u) \leftarrow true$
        for $w \in pred(u)$ do
            add $w$ to $pred(v)$
        done
        $u \leftarrow p_D(u)$
    done
```

Figure 3.13: Procedures that simulate the effect of tree path contractions.

3.3.3 Running time

Now we analyze the running time of PTRDOM assuming the one-phase *process_line*. (The same analysis — after some minor adjustments — is also valid for the two-phase version.)

Theorem 3.8 *Algorithm* PTRDOM *computes the immediate dominators of an input graph in* $O(n + m)$ *time on a pointer machine.*

Proof: The correctness of PTRDOM follows from Lemma 3.7 and [BKRW98b, Theorem 4.7]. In [BKRW98a] it is shown that *iidom*s can be computed in linear time on a pointer machine. The total time spent in *process_microtree* is linear plus the time spent on *vlink* and *veval*. As discussed earlier, after $O(|\ell|)$ preprocessing time, each call to RMQ(c, v) for any $c, v \in \ell$ takes constant amortized time. The total time spent on RMQ is thus

$$O(\sum_{\ell} |\ell| + m) = O(n + m).$$

Algorithm PTRDOM makes $O(n)$ calls to *vlink*. Now we count the number of calls to *veval*. The second phase of PTRDOM makes at most one call to *veval* per vertex. The first phase makes at most one call per arc in G, plus one call per arc in $A_\ell \setminus A$ for each line ℓ. Since each arc in $A_\ell \setminus A$ corresponds to an arc that enters $TL(\ell)$ in G, the total number of calls to *veval* is still $O(n + m)$. Each call takes constant time plus the time spent on *link* and *eval* for C'. The total time that will be spent on *link* and *eval* is

$$O(m\alpha(m, L) + L) = O(m),$$

for $g = O(\log^{1/3} n)$, since C' has $L = O(n/g)$ vertices. It remains to show that excluding the calls to *veval*, *process_line* takes linear time with respect to the number of vertices in ℓ and *TL*(ℓ) and the arcs entering them.

While ProcessLine processes a vertex v_j in $\ell = (v_1, \ldots, v_k)$ it examines all its predecessors in G plus the vertices added to the predecessors list of v_j, and then v_j is pushed onto S. If v_j is accessed again while processing v_i in ℓ then all the vertices $v_{j'}$, $i+1 \leq j' \leq j-1$, have been removed from S and are never accessed again. Thus the number of stack operations is proportional to the number of vertices in ℓ and their predecessors, and each stack operation takes constant time. Now consider a vertex u in *TL*(ℓ); *process_line* may visit u either because there is an arc (u, v_i) or an arc (u', v_i) for some proper descendant u' of u. The first time *process_line* visits u it spends time proportional to the number of predecessors of u because it copies them to the predecessors list of vertex v_i in ℓ and marks u. After that, u can be accessed again at most once for each child of u in D and once for each successor of u in *TL*(ℓ), and *process_line* spends $O(1)$ time for each such access. The Theorem follows because the total number of arcs added by *process_line* is less than the number of predecessors of the vertices in *TL*(ℓ). $\square$

3.4 Linear-time version of Lengauer-Tarjan

We present a modified version of algorithm PTRDOM, that computes for all vertices v the values $s(v)$ and $e(v) = \text{argmin}\{s(u) \mid s(v) \xrightarrow{+} u \xrightarrow{*} v\}$ in linear time. Hence, we essentially provide an implementation of the LT algorithm that runs in linear time and is simpler than the algorithm of Alstrup et al. [AHLT99] but slightly more complicated than the algorithm of Section 3.3. Algorithm PTRDOM computes precisely the $s(v)$ and $e(v)$ values for all vertices $v \in C$, but for the vertices u in the non-trivial microtrees it computes the values *pxdom*(u) and $\text{argmin}\{\textit{pxdom}(w) \mid \textit{pxdom}(u) \xrightarrow{+} w \xrightarrow{*} u\}$ instead. This suffices to compute the immediate dominators for all vertices, but other applications require the actual s and e functions. For example, in Section 5.4 we use these values to construct two spanning trees of G with a certain vertex-disjointness property.

In order to compute $e(v)$ for all v in non-trivial microtrees, it suffices to compute

$$e_T(v) = \operatorname{argmin}\{s(u) \mid \max\{s(v), r_T\} \xrightarrow{+} u \xrightarrow{*} v\},$$

where T is the non-trivial microtree containing v and r_T is its root. If $s(v) > r_T$, then $e_T(v) = e(v)$. Otherwise the computation can be completed by finding a vertex with minimum value on the path $C(s(v), p(r_T)]$; this can be done in Phase 2 of Algorithm PTRDOM by inserting r_T into the bucket associated with vertex $s(v)$.

We show how to compute the desired functions using external dominators (Section 3.2.1). Remember that the *xdom* values can be computed by running Phase 1 of Algorithm PTRDOM but with the call to *push_values* omitted in *process_microtree* and ignoring the call to *process_vertex*. This follows from [BKRW98b, Lemma 4.3], Lemma 3.7, and the fact that *xdom*$(v) = s(v)$ for all $v \in C$.

Figure 3.14 shows a slightly modified version of the LT algorithm, which we intend to use for the computation of s and e_T in each non-trivial microtree T. Note that *microLT* takes two additional parameters: a vertex r_T and a tree D. We assume that r_T is the root of the microtree T that we want to process. Then, *eval* operates only on paths that do not leave T. Also, D is the fixed DFS tree of G on which the s and e functions are defined. To get the original LT algorithm we set r_T equal to the root r of the flowgraph G, and D can be any DFS tree of G. Each vertex v is associated with an integer *value*(v), which initially equals v. In order to compute the desired values $s(v)$ and $e_T(v)$, we apply the algorithm to an augmented graph *xd*(T), which we define next.

Recall that $G(T)$ denotes the subgraph of G induced by the vertices of any microtree T. The augmented graph *xd*(T) contains one additional vertex for each distinct *xdom*(v) value x for $v \in T$ and $x \notin T$. Suppose $x_1 < \cdots < x_k$ are these *xdom* values, which are all less than r_T. Then *xd*(T) is constructed by adding to $G(T)$ the vertices $x_1, \ldots, x_k$ and the

```
$microLT(G = (V, A, r), D, r_T)$
    /* Phase 1 */
    for all $w \geq r_T$ in reverse preorder do
        for $v \in pred(w)$ do
            $u \leftarrow eval(v)$
            if $value(u) < value(w)$ then $value(w) \leftarrow value(u)$ endif.
        done
        $z \leftarrow \max\{r_T, value(w)\}$
        insert $w$ in $bucket[z]$
        $link(w)$
    done
    /* Phase 2 */
    re-initialize the link forest
    for all $w \geq r_T$ in reverse preorder do
        for all $v$ in $bucket[w]$ do
            delete $v$ from $bucket[w]$
            $e'_T(v) \leftarrow eval(v)$
        done
        $link(w)$
    done
```

Figure 3.14: Procedure *microLT* processes the graph induced by the vertices in D_{r_T} and their external dominators; D is a fixed DFS tree of G.

following arcs:

(i) (x_i, x_{i+1}), for $1 \leq i \leq k-1$,

(ii) (x_k, r_T), and

(iii) (x_i, v), for $1 \leq i \leq k$ and each $v \in T$ such that $xdom(v) = x_i$.

Finally we set x_1 to be the root of this flowgraph. We also need to specify a suitable DFS tree $D(T)$ of *xd*(T); this is formed from the path $(x_1, \dots, x_k)$ and T, by making x_k the parent of r_T. That is, $D(T)$ is formed by T and the arcs (i) and (ii). Now we show that *microLT* applied to *xd*(T) computes the semidominators of vertices in T.

Lemma 3.9 *Suppose we run algorithm microLT on xd*(T)*. Then after Phase 1 we will have value*$(v) = s(v)$ *for all* $v \in T$*.*

Proof: The Lemma is obvious if $s(v) \in T$, since for any vertex $w \geq v$ in an sdom path from $s(v)$ to v, we must have *xdom*$(u) = u > r_T$. Suppose $s(v) \xrightarrow{+} r_T$. Let $P = (s(v) = u_1, u_2, \dots, u_k = v)$ be an sdom path in G. Let u_i be the first vertex on P such that $u_i \in T$. By the definition of u_i the path $(u_1, \dots, u_{i-1})$ lies entirely outside T, and the properties of DFS imply that the path $(u_i, \dots, u_k)$ is entirely inside T. Because *xdom*$(u_i) = s(v)$, it follows that *value*$(v) = s(v)$ after processing v. □

The previous lemma immediately implies that after Phase 2, $e'_T(v) = e_T(v)$ for all $v \in T$.

Procedure *microLT* is a computation that depends only on the structure of the input graph; in particular the initial value of each v is itself, and no references are made outside individual augmented graphs. The size of each *xd*(T) is linear with respect to the size of $G(T)$, so *microLT* can be run on all the augmented graphs in linear time overall on a pointer machine. The details of these computations are similar to those of the computation of *iidom*s in [BKRW98a], but here we need to output two values ($s(v)$ and $e_T(v)$) per vertex instead of one.

It remains to show how to build the augmented graphs in linear time. In particular we have to show how to create efficiently the arcs of types (i) and (ii), since the construction of the arcs of type (iii) is straightforward during the computation of *xdom*s.

Arcs of type (i) must be constructed simultaneously for all microtrees, since two or more vertices in different microtrees may share the same *xdom*. To that end, as we compute the *xdom*s we maintain for each vertex $v \in C$ a list *xdlist* of pointers to all vertices w such that $\mathit{xdom}(w) = v$. After computing all the *xdom*s, we perform an extra pass over the vertices in C, in ascending DFS order. During this pass we maintain for each non-trivial microtree T a pointer p_T to the last *xdom* corresponding to T that was detected. At vertex v we traverse the associated *xdlist*. For each vertex u in that list, let $T = \mathit{micro}(u)$; we add the arc (p_T, v) to $\mathit{xd}(T)$ and set $p_T \leftarrow v$. Because the vertices that belong to the same microtree appear consecutively in *xdlist*, it is straightforward to guarantee that the arc (p_T, v) is added only once.

After completing the extra DFS traversal, we add arc (p_T, r_T) to $\mathit{xd}(T)$ for each non-trivial microtree T. This constructs the arcs of type (ii).

3.5 Without a linear-time NCA algorithm

The algorithm that we presented in Section 3.3 requires a linear-time solution to the off-line NCA problem in order to run in linear time. In this section we show that in fact we can use any naive method that computes nearest common ancestors on a fixed tree given the query set off-line.

For a given parameter γ that we will fix later, suppose we define a line of C to be a maximal unary path of size at most γ (previously $\gamma = n$). Let Λ be the number of lines, and L be the number of maximal unary paths of C as in Section 3.1 (previously $\Lambda = L$). A maximal unary path of length u_i is divided to at most $\lceil u_i/\gamma \rceil$ lines, therefore we have

$$\Lambda \leq \sum_{i=1}^{L} \left\lceil \frac{u_i}{\gamma} \right\rceil \leq \sum_{i=1}^{2n/g} \left(\frac{u_i}{\gamma} + 1 \right) \leq \frac{n}{\gamma} + \frac{2n}{g}.$$

We observe that our algorithm does not rely on the fact that a line is a maximal unary path in the core, but only on the fact that there are $o(n)$ lines. Therefore, as long as we can set γ and g so that $n/\gamma + 2n/g = o(n)$ the operations performed by the data structure of Section 3.3.1 will still run in linear time. Our goal is to perform some sort of simultaneous processing of every line, similar to the processing of the nontrivial microtrees by which the *iidom*s are computed. Hence, our first measure is to bound the size of the instances of the NCA problems that occur in our dominators algorithm, so that all possible distinct instances can be processed in linear time. We note that the Buchsbaum et al. off-line NCA algorithm achieves linear running time by partitioning the input tree into bottom-level microtrees of sufficiently small size. This means that Algorithm PTRDOM uses two types of microtrees: ones partitioning the DFS tree, and others partitioning the Cartesian trees. In essence our strategy is to divide the NCA problem into subproblems of small size explicitly, so that each Cartesian tree itself forms a microtree.

If we try to apply this idea in a straightforward manner, however, we encounter the following problem: Our algorithm requires answers to NCA queries (during the first phase) before all the semidominators in C are computed. This fact excludes the possibility of processing of every line simultaneously. For a RAM algorithm we can still use memoization, but on a pointer machine we have to apply some additional measures. A key observation is that the NCA queries on the Cartesian tree T_ℓ of line ℓ are performed only for the computation of the *pxdom*s in $\mathit{TL}(\ell)$; these values are not required for the computation of $\mathit{pxdom}(v)$ when $v \in \ell$ or $v > \mathit{bottom}(\ell)$; for $v < \mathit{top}(\ell)$ it suffices to have for all $u \in \mathit{TL}(\ell)$ values that satisfy the following equation:

$$\min\{\mathit{pxdom}(\mathit{up}(\eta(u))), \mathit{value}(u)\} = \min\{\mathit{pxdom}(\mathit{up}(\eta(u))), \mathit{pxdom}(u)\}. \quad (3.1)$$

We can compute such values for all $u \in \mathit{TL}(\ell)$ and for all lines ℓ without having to perform RMQs as follows. We rearrange the computations that are performed by our algorithm so that instead of using the value RMQ($\mathit{lastlinked}, v$) in *veval* we return $\mathit{value}(\mathit{up}(v))$ (Figure 3.8; see also Figure 3.9(c)). Then, after the corresponding microtree T has been processed, we will have

$$\textit{value}(u) = \min \{\textit{pxdom}(u), \textit{pxdom}(\textit{up}(\eta(u)))\},$$

for any $u \in T$, which implies equation (3.1). Of course, in order to compute $d(u)$ we need to compute *pxdom*(u) before the start of the idom phase. To that end, during the processing of u in the pxdom phase we can store a temporary value *temp*(u) for the "incomplete" *pxdom* calculations for u, which is the value that is computed by *process_microtree* if the results of all RMQs are suppressed and *push_values* is not executed. Then, the correct *pxdom*(u), for all $u \in T$, can be computed by setting *value*$(u) \leftarrow$ *temp*(u) and updating these values by preforming the suppressed RMQs (which can be done in a separate phase after the answers to all RMQs for all the lines are computed) and finally executing *push_values*.

After computing *pxdom*(v) for all $v \in C$ we can apply the Buchsbaum et al. algorithm for processing small graphs (Section 4 of [BKRW98a]) and answer all the pending NCAs in linear time, by choosing $\gamma = g = O(\log^{1/3} n)$. Here the small graphs that we process are the Cartesian trees T_ℓ corresponding to each line ℓ augmented with an edge $\{u, v\}$ for each query NCA$(T, \{u, v\})$.

3.6 Line dominators tree

In this section we present another alternative version of algorithm PTRDOM of Section 3.3, which is based on Lemma 2.6. This modified algorithm is actually more complex; the purpose of this section is to present some further results and properties of the structure of dominators computations.

Recall from Lemma 2.6 that $d(w) = \mathrm{NCA}(I, \{p(w), s(w)\})$. We used this lemma in the SEMI-NCA algorithm to construct I incrementally by observing that $d(w)$ is the nearest ancestor of $p(w)$ in I that is less or equal to $s(w)$. In the simple case where the DFS tree D is just a path this construction takes $O(n)$ time, because we never traverse an arc in I more than once.

```
$line\_dom(\ell = (v_1, \ldots, v_k))$

    $root \leftarrow parent_{I(\ell)}(v_1) \leftarrow s(v_1)$
    $child_{I(\ell)}(root) \leftarrow v_1$
    **for** $i \leftarrow 2, \ldots, k$ **do**
        $parent_{I(\ell)}(v_i) \leftarrow v_{i-1}$
        **if** $s(v_i) < root$ **then**
            $parent_{I(\ell)}(root) \leftarrow s(v_i)$
            $root \leftarrow s(v_i)$
            $child_{I(\ell)}(root) \leftarrow v_i$
            $parent_{I(\ell)}(v_i) \leftarrow root$
        **else**
            **while** $parent_{I(\ell)}(v_i) > s(v_i)$ **do**
                $parent_{I(\ell)}(v_i) \leftarrow parent_{I(\ell)}(parent_{I(\ell)}(v_i))$
            **done**
        **endif**
    **done**
```

Figure 3.15: Procedure *line_dom* builds the line dominators tree $I(\ell)$ of line ℓ.

Now we apply this method to a line $\ell = (v_1, v_2, \ldots, v_k)$. Suppose we have calculated the semidominators of the vertices in ℓ, so $value(v_i) = s(v_i)$, $1 \le i \le k$. Procedure *line_dom* of Figure 3.15 builds a tree $I(\ell)$ which we call the *line dominators tree* of ℓ, using the value $parent_{I(\ell)}(v)$ to point to the parent of v in $I(\ell)$. Figure 3.16 gives an example. The meaning of $I(\ell)$ is explained by Lemma 3.10.

Lemma 3.10 *Let $\ell = (v_1, \ldots, v_k)$. After the execution of line_dom(ℓ), let $u = parent_{I(\ell)} v_i$ for any $v_i \in \ell$. Then we have*

(a) $u = d(v_i) \in \ell$*, otherwise*

(b) $d(v_i) = d(z)$ *where* $z \in D(u, v_1)$ *and has minimum semidominator. Moreover if* $s(z) = u$ *then* $d(v_i) = u$.

Proof: If $v_1 = r$ then by Lemma 2.6, statement (a) is true for all v_i in ℓ. Now suppose

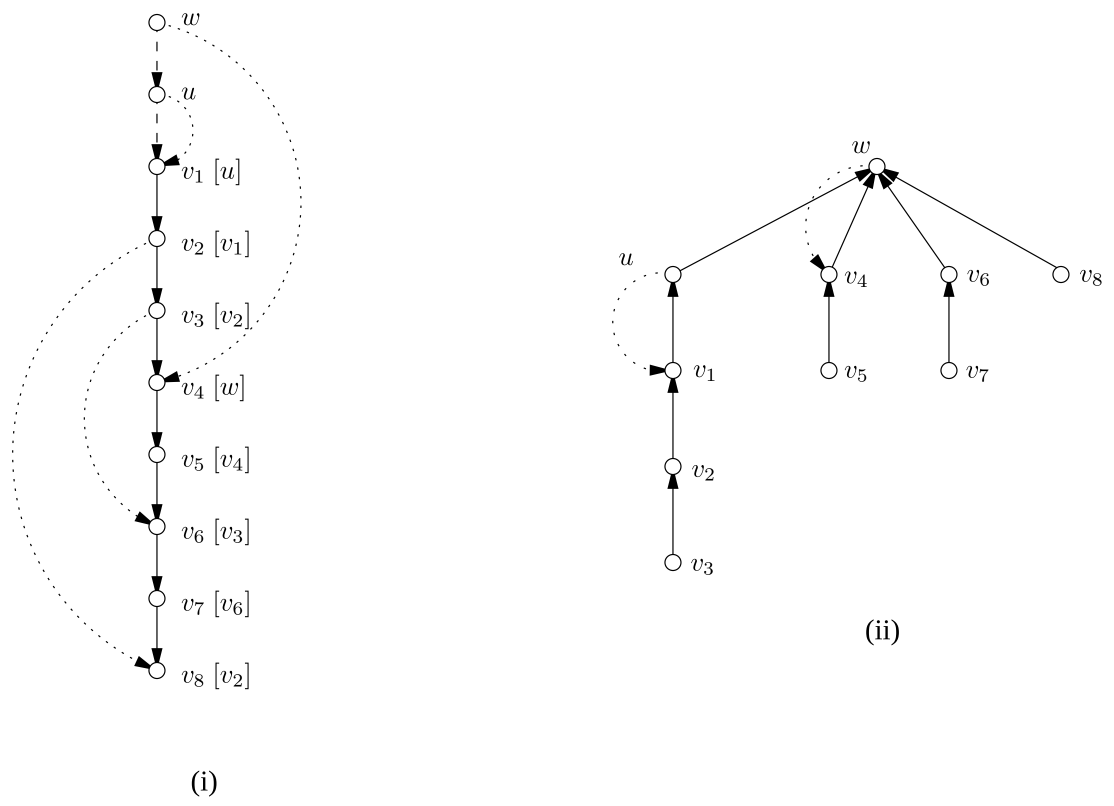


Figure 3.16: Example of the line dominators tree. (i) $\ell = (v_1, \dots, v_8)$ is a line of the core; w and u are semidominators of v_4 and v_1 respectively that lie outside ℓ. The dotted arcs connect $s(v_i) \neq p(v_i)$ to v_i and the values inside the brackets correspond to semidominators. (b) The line dominators tree $I(\ell)$. A solid arc (x, y) indicates that $\textit{parent}_{I(\ell)}(x) = y$. A dotted arc (y, x) indicates that $\textit{child}_{I(\ell)}(y) = x$, which means that during the process of constructing $I(\ell)$, y was once *root* and x become the first child of y in $I(\ell)$.

$v_1 \neq r$. We prove the Lemma by induction on i. For $i = 1$ statement (b) is true by Lemma 2.5. Suppose the Lemma is true for $1 \leq j \leq i-1$. If $v_{i'} = d(v_i) \in \ell$, then $1 \leq i' \leq i-1$ and by Lemma 2.2 we have that for any j such that $i'+1 \leq j \leq i-1$, $v_{i'} \xrightarrow{*} d(v_j)$. Therefore *line_dom* processes v_i as the second phase of SEMI-NCA and 2.6 implies that statement (a) is true.

Now assume $d(v_i) \notin \ell$. If $s(v_i) < \mathit{root}$ then $\mathit{parent}_{I(\ell)}(v_i) < \mathit{parent}_{I(\ell)}(v_j)$ for $1 \leq j < i$, so

$$\min\{s(z) \mid s(v_i) \xrightarrow{+} z \xrightarrow{*} v_i\} = \min\{s(z) \mid s(v_i) \xrightarrow{+} z \xrightarrow{+} v_1\},$$

and statement (2) is true by Lemma 2.5. If $s(v_i) > \mathit{root}$ then after v_i is processed we will have $\mathit{parent}_{I(\ell)}(v_i) = \mathit{root}$. Suppose that as v_i moves towards $root$ it visits the vertices $u_\lambda, u_{\lambda-1}, \ldots, u_1$, where $u_j \xrightarrow{+} u_{j+1}$ and $u_j > s(v_i)$, $1 \leq j \leq \lambda$. Also by induction hypothesis we have: $u_{j-1} = d(u_j)$ for $2 \leq j \leq \lambda$; u_1 is not dominated by any vertex in ℓ; and $d(u_1) \xrightarrow{*} \mathit{root}$. Lemma 2.2 implies $d(v_i) \xrightarrow{*} d(u_1)$. Suppose $d(v_i) \xrightarrow{+} d(u_1)$. Then there is a path P from $d(v_i)$ to v_i that avoids $d(u_1)$. Let $z \in P$ be the first vertex such that $d(u_1) \xrightarrow{+} z \xrightarrow{*} v_i$. If $z > u_1$ then $\mathit{root} > s(z)$, a contradiction. On the other hand z cannot be an ancestor of u_1 because this implies that there is a path from $d(v_i)$ to u_1 that avoids $d(u_1)$. Therefore, $d(v_i) = d(u_1)$. By the induction hypothesis statement (b) holds for u_1 so it must also hold for v_i. □

Note that all the vertices that point to the same vertex $u \notin \ell$ have the same relative dominator, which is $v = \mathit{child}_{I(\ell)}(u)$. Then it suffices to locate the immediate dominator of v only.

3.6.1 Modified *veval*

We use a modified version of *veval* which preforms NCA calculations in $I(\ell)$ instead of RMQs in the array $[s(v)]_{v \in \ell}$. Remember that we needed the value RMQ(v_{i+1}, v_j) when we process a vertex $u \in \mathit{TL}(v_i)$ (in the pxdom phase if we examine an arc (w, u) where $\eta(w) = v_j$, and in the idom phase if $v_i = \mathit{pxdom}(u)$). Now we will use the value of

NCA$(I(\ell), \{v_i, v_j\})$ instead. This is the only change we make to the pxdom phase (the idom phase requires further adjustments). We will show that the values we compute using our modified procedure give sufficient information to find the *pxdom*s of the vertices outside *TL*(ℓ), and the immediate dominators of vertices in *TL*(ℓ). First we will need the following result.

Lemma 3.11 *Let I be the dominator tree of $G = (V, A, r)$. Also let D be a DFS tree of G. Consider any vertices a and b, such that $a \xrightarrow{+} b$, and let $z \in D(a, b]$ be a vertex with minimum semidominator. Then $d(z) = \mathrm{NCA}(I, \{a, b\})$.*

Proof: By the definition of z we have $s(u) \geq s(z)$ for all $u \in D(a, b]$, and $s(z) \leq a$. Then, by Lemma 2.5, $d(z)$ dominates all vertices in $D[a, b]$. Suppose now we ascend the path on I from b to r until we reach the first vertex $x < z$. For any vertex $u \in D(z, b]$, Lemma 2.2 implies either $z \xrightarrow{*} d(u)$ or $d(u) = d(z)$. Therefore, we have $x = d(z)$. But since $d(z) \in$ *dom*(a) and $a < z$, x is the closest common dominator of a and b. □

Now our goal is to show that for any $v \in V$, the vertex *value*(v) computed in the pxdom phase satisfies the following lemma.

Lemma 3.12 *Let T be any microtree and let u be any vertex of T. Then, after the execution of process_microtree for T, Equation* (3.1) *holds. Moreover, if T is not a left microtree then value(u) = pxdom(u). Otherwise if $d(u) \notin T$, value(u) satisfies one of the following:*

(a) value(u) = pxdom(u).

(b) value$(u) = d(u)$.

(c) $d(u) = d(x)$, where $x \in D($value$(u), \eta(u)]$ is a vertex with minimum semidominator.

Proof: The proof is a tedious induction on the order the microtrees are processed. For the basis of the induction we consider the first microtree T that was processed. The Lemma follows immediately for T; no NCA computations are involved during the execution of *process_microtree* because T cannot be a non-trivial microtree to the left of a line, and thus *value*$(u) =$ *pxdom*(u). The same argument, in conjunction with the induction hypothesis, implies that we will have *value*$(u) =$ *pxdom*(u) for all u such that $u \in C$ or *micro*(u) is not a left microtree. Hence, for the induction step we only need to consider a non-trivial microtree T to the left of a line ℓ.

First we will show that just before PushValues(T) we have for any $v \in T$, *value*$(v) \leq$ *xdom*(v); this follows by the induction hypothesis, since *value*$(u) \leq$ *pxdom*(u) for any $u \notin T$ such that $u > v$, and the fact that for any line $\ell' = (v_1, \ldots, v_i, \ldots, v_j, \ldots, v_k)$,

$$\mathrm{NCA}(I(\ell'), \{v_i, v_j\}) \leq \mathrm{RMQ}(v_{i+1}, v_j).$$

This also implies that after *push_values*(T) we have *value*$(v) \leq$ *pxdom*(v). We prove the Lemma in two steps. First we show that the Lemma holds for any vertex $u \in T$ such that *value*(u) did not change after *push_values*(T). Then, we show that the Lemma also holds for any vertex $u' \in T$ such that *value*(u') was pushed by some other vertex $u \in T$.

Case 1: *value*(u) **did not decrease after** *push_values*(T)**.** We consider which case of Figure 3.9 applied when *value*(u) got its minimal value. Let x be the predecessor of u that caused the corresponding computation. Consider case (a) when $\eta(x) = r_F(\eta(x))$. Then we have *value*$(u) =$ *pxdom*(x) if $x \notin C$, and *value*$(u) = x$ otherwise. This implies *value*$(u) \geq$ *pxdom*(u). But we have already shown that *value*$(u) \leq$ *pxdom*(u), thus *value*$(u) =$ *pxdom*(u) and the Lemma follows. In cases (a) when $r_F(\eta(x)) \xrightarrow{+} \eta(x)$, and (b), we have:

$$\mathit{value}(u) = \min\Big(\{\mathit{value}(x)\} \cup \Big\{\mathit{pxdom}(z) \mid r_F(\eta(x)) \xrightarrow{+} z \xrightarrow{*} \eta(x)\Big\} \Big).$$

By induction hypothesis (3.1) holds for x which means

$$\min\big(\{\mathit{value}(x), \mathit{pxdom}(\mathit{up}(\eta(x)))\}\big) = \min\big(\{\mathit{pxdom}(x), \mathit{pxdom}(\mathit{up}(\eta(x)))\}\big),$$

so in both cases *value*$(u) =$ *pxdom*(u).

In case (a), *value*(u) was either the result of the NCA calculation or *value*$(u) =$ *value*(x).

First consider the case

$$\textit{value}(u) = \mathrm{NCA}(I(\ell), \{v_i, v_j\}), \tag{3.2}$$

where $v_i = \eta(u)$ and $v_j = \eta(w)$ for some $w \in \textit{pred}(u)$. Then T belongs to $\textit{TL}(\ell)$. Let $r_i = \textit{root}(T)$, and let $v_1 = \textit{top}(\ell)$. If $d(u) \in T$ then we must have $s(x) \geq v_i$ for any $x \in D(v_i, v_j]$, which implies $\textit{value}(u) = v_i$. But then $\textit{pxdom}(u) = v_i = \textit{value}(v)$ and Equation (3.1) and Statement (a) hold. Hence, $\textit{value}(u) < \textit{pxdom}(u)$ implies $d(v) \notin T$. Let $z \in D(v_i, v_j]$ be a vertex of minimal semidominator. Then we have

$$s(u) \leq s(z), \tag{3.3}$$

$$s(y) \geq \textit{value}(u),\ r_i \xrightarrow{*} y \xrightarrow{*} u, \tag{3.4}$$

$$s(y) \geq s(z),\ v_i \xrightarrow{+} y \xrightarrow{*} v_j, \tag{3.5}$$

$$s(y) \geq \textit{value}(u),\ \max\{\textit{value}(u), p(v_1)\} \xrightarrow{+} y \xrightarrow{*} v_i. \tag{3.6}$$

Inequality (3.3) holds because any sdom path to z can be extended to an sdom path to u using $D[z, v_j] \cdot D[v_j, w] \cdot (w, u)$; (3.4) follows from the assumption that $\textit{value}(u)$ did not decrease after $\textit{push\_values}(T)$; (3.5) is implied by the definition of z; finally (3.6) follows from (3.2). By Lemma 2.6, $d(z) = \mathrm{NCA}(I, \{s(z), p(z)\})$. Clearly

$$\mathrm{NCA}(I, \{s(z), p(z)\}) \leq \mathrm{NCA}(I(\ell), \{s(z), p(z)\})$$

with the equality holding if $\mathrm{NCA}(I(\ell), \{s(z), p(z)\}) \in \ell$. Also Lemma 3.10 and Lemma 3.11 imply

$$\mathrm{NCA}(I(\ell), \{s(z), p(z)\}) = \mathrm{NCA}(I(\ell), \{v_i, v_j\}),$$

so

$$\textit{value}(u) = \mathrm{NCA}(I(\ell), \{s(z), p(z)\}), \tag{3.7}$$

and $\textit{value}(u) \geq d(z)$ For each vertex $y \in D(d(z), z)$ there is a path P from $d(z)$ to z that avoids y. Therefore $P \cdot D[z, v_j] \cdot D[v_j, w] \cdot (w, u)$ avoids y, and therefore $d(u) \leq d(z)$. But inequality (3.4) and the fact $d(z) \leq \textit{value}(u)$ exclude the possibility that $d(u) < d(z)$, hence $d(u) = d(z)$. Now if $\mathit{value}(u) \in \ell$ then

$$\min\{\mathit{value}(u), \mathit{pxdom}(\mathit{up}(v_i))\} = \mathit{pxdom}(\mathit{up}(v_i)) = \min\{\mathit{pxdom}(u), \mathit{pxdom}(\mathit{up}(v_i))\},$$

and (3.1) holds. By Lemma 2.6 for z we have

$$\mathit{value}(u) = \mathrm{NCA}(I(\ell), \{s(z), p(z)\}) = d(z),$$

which means $\mathit{value}(u) = d(u)$ and statement (b) follows. Finally consider $\mathit{value}(u) \notin \ell$. Notice that if $s(z) < v_1$ and $s(y) \geq s(z)$ for $y \in D[v_1, v_i]$ then $\mathit{value}(u) = s(z)$, so $\mathit{value}(u) = \mathit{pxdom}(u)$. Otherwise (when $s(z) > v_1$ or $s(y) < s(z)$ for some $y \in D[v_1, v_i]$) $\mathit{value}(u) \geq \mathit{pxdom}(\mathit{up}(v_i))$ and $\mathit{pxdom}(u) \geq \mathit{pxdom}(\mathit{up}(v_i))$. Then in any case, (3.1) holds. Also, by equation (3.7) and Lemma 3.10 for z, we have that one of the statements (a), (b) and (c) must hold for u.

Now assume $\mathit{value}(u) = \mathit{value}(x)$. Notice that if $\mathit{value}(x) = \mathit{pxdom}(x)$ then $\mathit{value}(u) = \mathit{pxdom}(u)$ and the Lemma holds. Otherwise we have $\mathit{value}(x) < \mathit{pxdom}(x)$, thus $x \notin C$. Also by induction hypothesis and (3.1), we have $\mathit{value}(x) \geq \mathit{pxdom}(\mathit{up}(v_j))$. Since $v_i \xrightarrow{+} v_j$, $\mathit{pxdom}(\mathit{up}(v_i)) \geq \mathit{pxdom}(\mathit{up}(v_j))$. Clearly if $\mathit{pxdom}(\mathit{up}(v_i)) = \mathit{pxdom}(\mathit{up}(v_j))$ then (3.1) holds for u. Otherwise $\mathit{pxdom}(\mathit{up}(v_i)) > \mathit{pxdom}(\mathit{up}(v_j))$ and there is a pxdom path from $\mathit{pxdom}(\mathit{up}(v_j))$ to u. Hence, $\mathit{pxdom}(u) \leq \mathit{pxdom}(\mathit{up}(v_j))$ which implies $\mathit{value}(u) \leq \mathit{pxdom}(\mathit{up}(v_j))$. But $\mathit{value}(x) \geq \mathit{pxdom}(\mathit{up}(v_j))$ so we must have $\mathit{value}(u) = \mathit{pxdom}(u) = \mathit{pxdom}(\mathit{up}(v_j))$ and (3.1) follows. We proceed to show that on of the Statements (a), (b) and (c) holds when $d(u) \notin T$. Note that $d(u) \notin T$ implies $d(u) \xrightarrow{*} d(x)$. Suppose $d(u) \xrightarrow{+} d(x)$, and let P be a path from $d(u)$ to u that avoids $d(x)$. This path does not contain any vertex $w \in D(d(x), x]$. Otherwise x would not be dominated by $d(x)$. But then we have $\mathit{pxdom}(u) < d(x)$, and thus, $\mathit{value}(x) = \mathit{value}(u) \leq \mathit{pxdom}(u) < d(x)$, which contradicts the induction hypothesis $\mathit{value}(x) \geq d(x)$. Thus, $d(u) = d(x)$. Clearly $\mathit{value}(x) < v_j$, so $d(x) \notin \mathit{micro}(x)$. Then one of the statements (a)-(c) holds for x, hence the same statement must also hold for u.

Case 2: *value*(u') **decreased after** *push_values*(T)**.** Now we prove the Lemma for any vertex $u' \in T$, where u' is such that *value*(u') was pushed from some other vertex $u \in T$. This means *value*$(u') >$ *value*(u) before *push_values*(T) and *value*(u) did not decrease after *push_values*(T). Then there is a path P_T from u to u' in the graph $G(T)$ induced by the vertices of T, and *value*$(x) =$ *value*(u) after *push_values*(T) for all $x \in P_T$. Clearly *pxdom*$(u') \leq$ *pxdom*(u). Note that if *value*$(u) =$ *pxdom*(u) then we must have *value*$(u') =$ *pxdom*(u'), and the Lemma is true. Now assume *value*$(u) <$ *pxdom*(u). If *value*(u) is in ℓ then *pxdom*(u') is also in ℓ and we must have *pxdom*$(u') >$ *pxdom*$(up(v_i))$. Otherwise the assumption *value*$(v) <$ *pxdom*(v) and (3.1) imply *value*$(v) =$ *pxdom*$(up(v_i))$. So *value*$(u') =$ *pxdom*$(up(v_i)) \leq$ *pxdom*(u'), which means

$$\min\left(\{\mathit{value}(u'), \mathit{pxdom}(up(v_i))\}\right) = \mathit{pxdom}(up(v_i)) = \min\left(\{\mathit{pxdom}(u'), \mathit{pxdom}(up(v_i))\}\right)$$

and (3.1) holds for u'. We proceed to statements (a)-(c) assuming $d(u') \notin T$. Note that $d(u) \notin T$ and $d(u') \xrightarrow{*} d(u)$. If $d(u') \xrightarrow{+} d(u)$ then there is a path P from $d(u')$ to u' that avoids $d(u)$. If P contains a vertex $w \in D(d(u), u]$ then we have a path from $d(u')$ to u that avoids $d(u)$, a contradiction. Hence, P does not intersect $D(d(u), u]$. But again this is impossible since it implies *pxdom*$(u') < d(u)$ and we know that *value*$(u) \geq d(u)$ because Case 1 holds for u. Therefore $d(u) = d(u')$ and the result follows by induction hypothesis.
□

3.6.2 Immediate dominators

The following lemma is analogous to Lemma 2.6.

Lemma 3.13 *Consider a line $\ell = (v_1, v_2, \ldots, v_k)$ and let $v \neq v_i$ be any vertex in either TL(v_i) or TR(v_i). If $d(v)$ is not in $T =$ micro(v) then $d(v) = \mathrm{NCA}(I, \{v_i, pxdom(v)\})$.*

Proof: By Lemma 3.1 there is an ancestor $w \in T$ of v such that *pxdom*$(v) =$ *pxdom*$(w) = s(w)$ and $d(w) \notin T$. By assumption $d(v) \notin T$ so the same Lemma implies $d(v) = d(w)$. By

Lemma 2.6 we have

$$d(w) = \mathrm{NCA}(I, \{s(w), p(w)\}) = \mathrm{NCA}(I, \{\mathit{pxdom}(w), p(w)\}).$$

Note that for any vertex $u \in D(v_i, w]$, $s(u) \geq s(w)$. Let $z \in D(s(w), v_i]$ be a vertex with minimum semidominator. Then by Lemma 3.11 we have $d(z) = \mathrm{NCA}(I, \{s(w), v_i\})$. But $d(z) = d(w)$, by Lemma 2.5 so the Lemma follows. □

In the previous proof if we apply Lemma 3.10 instead of Lemma 2.6 we get the following result.

Lemma 3.14 *Consider a line $\ell = (v_1, v_2, \ldots, v_k)$ and let $v \neq v_i$ be any vertex in either TL(v_i) or TR(v_i). Let*

$$y = \mathrm{NCA}(I(\ell), \{v_i, \mathit{pxdom}(v)\}).$$

If $d(v)$ is not in $T = \mathit{micro}(v)$ then one of the following statements is true:

(a) $y = idom(v) \in l$, otherwise

(b) $d(v) = d(z)$, where $z \in D(y, v_1)$ is a vertex with minimum semidominator. Moreover if $s(z) = y$ then $d(v) = y$.

In order to complete the description of our modified algorithm we now give the details of computing the immediate dominators. We use a modified version of *process_bucket* that works as follows. Consider a vertex v, such that $\eta(v) \in \ell$. At the end of the pxdom phase we insert v into a bucket associated with a vertex u that we define next.

First suppose $v \in C$. Then we set $u \leftarrow \mathit{parent}_{I(\ell)}v$. If $u \in \ell$ then $d(v) = u$ and we are done. Otherwise, we need to find a vertex $z \in D(u, v_1)$ with minimum value. Note that $u \leq s(z)$ for any $z \in D[v_1, v]$. Hence, *veval*(v) gives the desired result; it evaluates the minimum value on the path $D(u, v]$ which extends beyond ℓ, so it does not perform any RMQ. By applying Lemma 3.10 this computation returns either the immediate dominator or a relative dominator of v.

Now suppose $v \notin C$. Then we set $u \leftarrow \mathrm{NCA}(I(\ell), \{\eta(v), \mathit{value}(\eta(v))\})$. We consider the cases listed in Lemma 3.12. If $\mathit{value}(v) = \mathit{pxdom}(v)$ then, by Lemma 3.14, $u \geq v_1$

```
process_bucket(bucket[u])

    for $v \in bucket[u]$ do
        if $u \geq top(line(\eta(v)))$ then
            $idom[v] \leftarrow u$
            return
        endif
        $z \leftarrow veval(\eta(v))$
        if $s(z) = u$ then $idom[v] \leftarrow u$ else $idom[v] = idom[z]$ endif
    done
```

Figure 3.17: The modified processing of a bucket.

implies $d(v) = u$. Otherwise, we compute *veval*$(\eta(v))$ which returns either the immediate dominator of v or a relative dominator. Finally suppose *value*$(v) \neq$ *pxdom*(v). By Lemma 3.12, $u \geq v_1$ again implies $d(v) = u$. If $u < v_1$, *veval*$(\eta(v))$ returns either the immediate dominator or a relative dominator of v.

Figure 3.17 gives an implementation of the modified routine that processes *bucket*$[u]$.

3.6.3 Remarks

The computations performed by the algorithm we presented in this section are equivalent to the computations of Section 3.3 but the modified algorithm is conceptually more complicated. Another drawback of the modified algorithm is that a line dominators tree $I(\ell)$ may have $2\,|\ell|$ vertices, i.e., twice as many as the Cartesian tree T_ℓ. On the other hand $I(\ell)$ gives a closer approximation to the immediate dominators than T_ℓ, which may lead to fewer computations during the idom phase, as discussed in Section 3.6.2.

Interestingly, the techniques we presented in this section allow us to compute during the idom phase the immediate dominators of all the vertices in the core *without any* off-line (RMQ or NCA) computations. Then the immediate dominators of the vertices in non-trivial microtrees can be found by off-line NCA calculations on the dominator tree of the core.

Chapter 4

Dominator Tree Verification

The relationship of verification to computation is highly problem-dependent. Consider, for example, the situation for shortest path trees and for minimum spanning trees. There is a very simple linear-time algorithm to verify a shortest path tree, but no linear-time algorithm to compute shortest path trees is known (for a comparison-based computation model). On the other hand, minimum spanning trees can be verified in linear time [DRT92, Kin97, BKRW98a], but the known methods are complicated. By combining a linear-time verification algorithm with random sampling, one can actually find a minimum spanning tree in linear time [KKT95].

In this chapter we study the problem of verifying a dominator tree: Given a flowgraph G and a tree T, we want to decide whether T is the dominator tree of G. We present a linear-time algorithm to verify a dominator tree. This algorithm is simpler than the known linear-time algorithms to find dominator trees, but also non-trivial. An $O(m\alpha(m,n))$-time version of our algorithm is simpler than the $O(m\alpha(m,n))$-time algorithm for finding dominators; it requires only a standard set union data structure instead of a *link-eval* data structure [Tar79a] (see also Section 2.2.2). Our work sheds light on the relationship between verification and computation of dominators, and we hope it will lead to a simpler linear-time algorithm to find dominators.

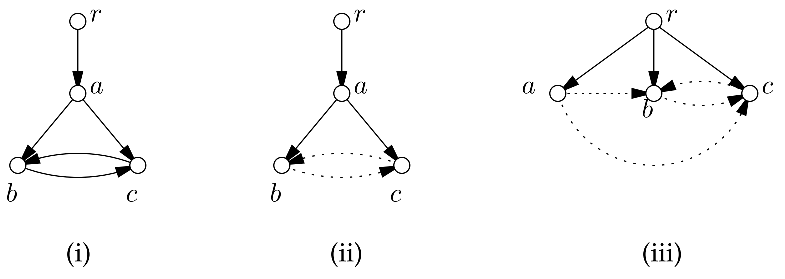


Figure 4.1: Condition (4.1) is not sufficient for graphs with cycles. (i) A flowgraph G with 4 vertices. (ii) The dominator tree I of G; the dotted arcs are in $G - I$. (iii) A different tree T that satisfies Condition (4.1); the dotted arcs are in $G - T$.

4.1 Necessary condition.

It is known that the proposed dominator tree T must satisfy the following condition [RR94]:

$$p_T(w) = \mathrm{NCA}(T, pred(w)), \ \forall\, w \in V - r. \tag{4.1}$$

This condition is not sufficient in general but it is sufficient for acyclic graphs. Figure 4.1 shows a counter-example for a graph that contains a cycle. Nonetheless, Condition (4.1) provides some useful information about the location of the dominators of each vertex, as we show in the next lemma.

Lemma 4.1 *Let T be a tree that satisfies (4.1). Then for any $w \neq r$, $p_T(w)$ dominates w.*

Proof: Let $u = p_T(w)$. Since T satisfies (4.1), for any z in $T_u - u$ we have $pred(z) \subseteq T_u$. Therefore any path from r to w must pass through u. Also since w is reachable from r there must be at least one such path, so u dominates w. □

Corollary 4.2 *Let T be a tree that satisfies (4.1). Then, for any vertex $w \neq r$, $d(w)$ is a descendant of $p_T(w)$ in T. Moreover, if $p_T(w) \in pred(w)$ then $p_T(w) = d(w)$.*

Hence, in a tree that satisfies Condition (4.1), the vertices on a path from the root to any vertex v are dominators of v, although they may comprise only a subset of $dom(v)$. Then if G contains only trivial dominators, Condition (4.1) is satisfied by a unique tree;

every $v \in V - r$ is a child of the root. The main idea of our verification algorithm is based on the previous observation, and is as follows. Given G and T we construct a graph G_T that satisfies the following property:

$$T \text{ is the dominator tree of } G \Longleftrightarrow \forall\, v \;\; G_T(v) \text{ contains only trivial dominators.}$$

For any vertex v, $G_T(v)$ is the subgraph of G_T that is induced by v and its children in T. (If v is a leaf in T then $G_T(v)$ consists only of v.) Thus we reduce the verification problem to the problem of testing if a graph contains only trivial dominators.

Our reduction is based on the idea of the *derived graph*, which we denote by G_T and which was introduced in [Tar81]. We give the definition of the derived graph together with a linear-time procedure to construct it in the following section.

4.2 Derived graph.

Let $G_T = (V, A_T, r)$. The arc set A_T is formed by the map $\gamma : A \mapsto A_T \cup \{\textit{null}\}$, defined as:

$$\gamma(w,v) = \begin{cases} (w,v), & w = p_T(v) \\ (u,v), & u \neq v,\ p_T(u) = p_T(v) \text{ and } u \xrightarrow{*}_T w \\ \textit{null}, & \text{otherwise} \end{cases} . \tag{4.2}$$

Note that γ is a many-to-one map, and each arc in A_T may correspond to several arcs in A. Any arc in A mapped to *null* is discarded. In particular we ignore any arc $(w,v) \in A$ such that $v \xrightarrow{*}_T w$; this arc does not contribute any dominance information since all the vertices on $T[v,w]$ should be dominated by v. Our definition also excludes any arc $(w,v) \in A$ such that

$$\text{NCA}(T, \{w,v\}) \neq p_T(v), \text{ for } v \overset{*}{\not\to}_T w.$$

Note that if such an arc exists then T does not satisfy the necessary Condition (4.1). In this case our algorithm reports that the input tree is not the dominator tree of G and terminates. The remaining arcs of G are mapped to two kinds of arcs in G_T:

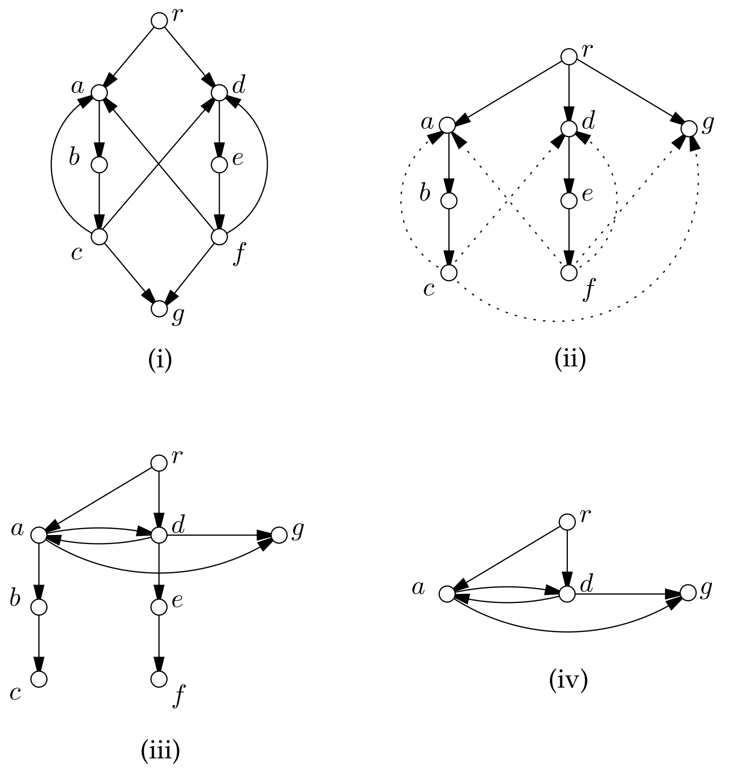


Figure 4.2: Derived graph and subgraphs. (i) The flowgraph $G = (V, A, r)$. (ii) The proposed dominator tree T. In this example $T = I$. Non-tree arcs are dotted. (iii) The derived flowgraph $G_T = (V, A_T, r)$. The arcs (c, a) and (f, d) of G are eliminated. The arcs (c, d) and (c, g) of G correspond to (a, d) and (a, g) in G_T. The arcs (f, a) and (f, g) of G correspond to (d, a) and (d, g) in G_T. (iv) The derived subgraph $G_T(r)$.

- arcs that lead from a parent to a child in T, and

- arcs that lead from a vertex to one of its siblings in T.

For any vertex w we define the *derived subgraph* $G_T(w)$ to be the subgraph of G_T induced by w and its children in T. Figure 4.2 gives an example of these definitions.

The next lemma shows that it suffices to verify that T is the dominator tree of G_T.

Lemma 4.3 *Let T be a tree that satisfies (4.1). Then T is the dominator tree of G if and only if T is the dominator tree of G_T.*

Proof: Let w be any vertex of G other than r and let $u = p_T(w)$. We have assumed that T satisfies Condition (4.1) for G, which implies that the same condition holds for G_T. Let $d_G(w) \equiv d(w)$ and $d_{G_T}(w)$ be respectively the immediate dominators of w in G and in G_T. By Corollary 4.2 we have that both $d_{G_T}(w)$ and $d_G(w)$ are in T_u. If u is a predecessor of w then u is the immediate dominator of w in both G and G_T.

Now assume that $u = d_G(w)$ but $(u, w) \notin A$. Then there exists a simple path $P = (v_0 = u, v_1, \dots, v_k, v_{k+1} = w)$ in G. Also, by the definition of $d_G(w)$, for each i in $[1, k]$ there exists a simple path P_{v_i} in G from u to w that avoids v_i. Condition (4.1) implies that $v_i \in T_u$, $0 \le i \le k+1$. First we show that each path from u to w in G induces a path from u to w in G_T. To that end we assume that $chd_T(u) = (u_1, u_2, \dots, u_d)$ and show that P induces a path P' in G_T that also connects u to w. Note that w and v_1 are both in $chd_T(u)$, so P contains some vertices of $chd_T(u)$. Furthermore we can partition P into subpaths

$$P[u_s, u_t] = (x_0 = u_s, x_1, \dots, x_l, x_{l+1} = u_t),$$

such that $x_i \notin chd_T(u)$, $1 \le i \le l$. Then since T satisfies Condition (4.1) we have $x_i \in T_{u_s}$, $1 \le i \le l$. This means $\gamma(x_l, u_t) = (u_s, u_t)$ and A_T contains the arc (u_s, u_t) that corresponds to the arc $(x_l, u_t) \in A$. Therefore the subpath $P[u_s, u_t]$ in G is translated to the arc (u_s, u_t) in G_T and we conclude that P is translated to a path $P' = (u, u_{i_1}, \dots, u_{i_j}, w)$ in G_T, where $u_{i_{j'}} \in P$, $1 \le j' \le j$. By the same arguments $P_{u_{i_{j'}}}$ is translated to a path $P'_{u_{i_{j'}}}$ in G_T that avoids $u_{i_{j'}}$. This proves that $u = d_G(w)$ implies $u = d_{G_T}(w)$.

Now we consider the case $u \neq d_G(w)$. By Corollary 4.2 we have $z = d_G(w) \in T_u - u$. Let u_i be the child of u that is an ancestor of z. Every path P from u to w in G contains z, therefore the induced path P' in G_T must contain u_i. Moreover, as we argued before, for any $u_j \neq u_i$ in P the induced path P'_{u_j} in G_T avoids u_j. We conclude that $u_i = d_{G_T}(w)$. □

Now we describe an algorithm that builds G_T and at the same time verifies that T satisfies Condition (4.1). First we note that an equivalent way to state Condition (4.1) is that for each $x \neq r$ the following two properties must hold:

(P1) For all z in *pred*(x), $p_T(x) \overset{*}{\rightarrow}_T z$.

(P2) Either $p_T(x)$ is in *pred*(x) or there exist two distinct children of $p_T(x)$ in T, y_1 and y_2, and predecessors of x, z_1 and z_2, such that $y_i \neq x$ and $y_i \overset{*}{\rightarrow}_T z_i$, for $i = 1, 2$.

We start by constructing for each vertex v that is not a leaf in T the list of its children in T, *chd*$_T(v)$. Then we perform a preorder walk of T. The first time we visit a vertex v we assign to it a preorder number and the last time (after visiting all the vertices in T_v) we compute the size of the subtree rooted at v; to do so, we assign

$$\mathit{size}(v) \leftarrow \sum_{u \in \mathit{chd}_T(v)} \mathit{size}(u) \ + \ 1.$$

For simplicity, we will refer to the vertices of T by their preorder numbers. Then $v < u$ means that v was visited before u during the preorder walk. Since the vertices of a subtree are assigned consecutive numbers, $v \overset{*}{\rightarrow}_T u$ if and only if $v \leq u \leq v + \mathit{size}(v) - 1$. So, Property (P1) can be tested in constant time per edge. In order to test Property (P2) we need some additional information, which can also be collected during the preorder walk of T. When we visit a vertex v we examine its list of successors. Let $u \in \mathit{succ}(v)$ and let $w = p_T(u)$. If $v \neq w$ and $u \overset{*}{\not\rightarrow}_T v$, then we insert v at the end of a list *predom*(w). This list stores the predecessors v of the vertices u for which we want to verify that w is their immediate dominator. Figure 4.3 illustrates these definitions. Notice that since we visit the vertices in preorder, *predom*(w) is sorted in ascending order. Moreover the list that represents *chd*$_T(w) = (w_1, w_2, \ldots, w_k)$ is also sorted with respect to the preorder numbering if we visit the children of v in the order given by the list. For each vertex $v \in$ *predom*(w) we want to find the child of w that is an ancestor of v. Let $a_T(w, v)$ be that child of w. Since the lists that represent both *chd*$_T(w)$ and *predom*(w) are sorted in ascending order, we can find all $a_T(w, v)$ in time

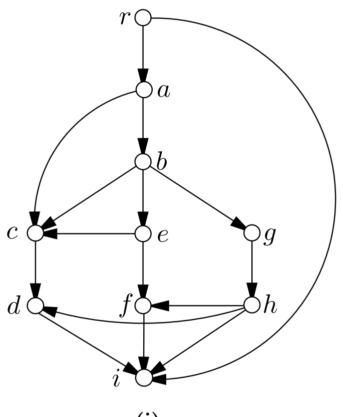
$$O\big(\,|\mathit{predom}(w)| + |\mathit{chd}_T(w)|\,\big).$$

Clearly $a_T(w, v)$ is the vertex w_i in the list *chd*$_T(w)$ that satisfies $w_i \leq v$ and $w_{i+1} > v$ if $1 \leq i < k$, or $w_k \leq v \leq w + \mathit{size}(w) - 1$. (If v does not satisfy any of these inequalities then Property (P1) does not hold.) Thus, we are essentially merging the two sorted lists that represent *chd*$_T(w)$ and *predom*(w). Furthermore, a vertex v can occur in at most $|\mathit{succ}(v)|$ *predom* lists. Hence, the total time that will take us to compute $a_T(w, v)$ for all $w \in T$ and all $v \in$ *predom*(w) is proportional to

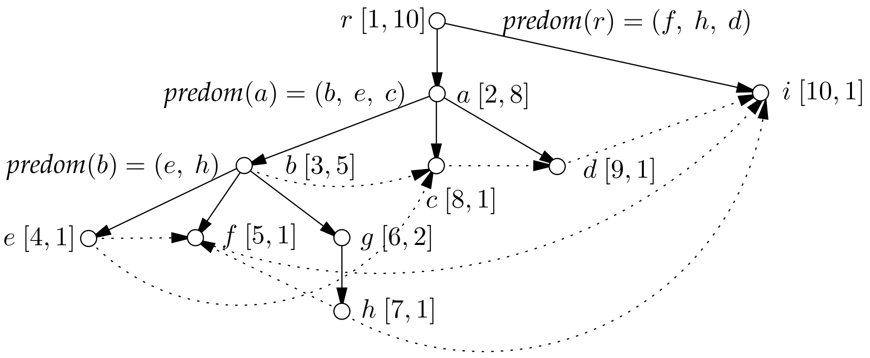
$$\sum_{w \in T} \big(\,|\mathit{predom}(w)| + |\mathit{chd}_T(w)|\,\big) \leq |A| + |V| = m + n.$$

After calculating $a_T(w, v)$ for each $v \in$ *pred*(u) such that $w = p_T(u)$, testing Property (P2) takes linear time.

As we mentioned earlier, if a test for (P1) or (P2) fails for some vertex, then our algorithm reports that $T \neq I$ and terminates. Otherwise, it uses the $a_T(w, v)$ information to construct the derived graph. The arcs $(p_T(v), v)$ in A are copied to A_T. For any other arc (u, v) in A, we include in A_T the corresponding arc $\gamma(u, v) = (z, v)$ where $z = a_T(p_T(v), u)$. Note that if we store the $a_T(w, v)$ values as we compute them in a linked list associated with v (which will be the list of predecessors of v in G_T), then these values will be sorted in ascending order because the vertices are visited in preorder in T. This fact enables us to avoid introducing multiple arcs in G_T.



(i)



(ii)

Figure 4.3: Construction of the derived graph. (i) The input flowgraph. (ii) The proposed dominators tree T. In this example $T = I$. Dotted arcs are not in T. The values inside the brackets correspond to the preorder number of a vertex and the size of its subtree. Also the nonempty *predom* lists are shown.

4.3 Acyclic graphs.

As we mentioned earlier, Condition (4.1) is necessary and sufficient for acyclic graphs. Hence, the verification procedure can accept T if it successfully completes the tests for (P1) and (P2) at each vertex.

4.4 Reducible graphs.

If G is reducible[1] then for every back arc (u, v), with respect to a fixed depth-first search tree D of G, we have that v dominates u. Such arcs do not contribute any dominance information, and hence can be removed. The resulting graph is acyclic and has the same dominators as G, so we can apply the same verification procedure as for acyclic graphs.

4.5 General graphs.

In the general case, for each derived sub-flowgraph $G_T(w) = (V_T(w), A_T(w), w)$, we have to check if all the vertices in $G_T(w)$ have only trivial dominators. If this is true then Corollary 4.2 implies that $d_{G_T}(v) = w$ for all $v \in V_T(w) - w$, and by Lemma 4.3 we have $d(v) = w$. In the next section we see how to verify in linear time that a graph has only trivial dominators.

4.5.1 Verifying trivial dominators.

We will describe a subroutine that given a flowgraph $G = (V, E, r)$ checks whether r is the immediate dominator of every vertex $v \in V - r$. Initially we perform a DFS on G, which produces a DFS tree D and a preorder numbering for the vertices of G. We refer to the vertices by their preorder numbers in D, so $v < u$ means that v was visited before u during the DFS. Also we define $\nu(u, v)$ to be an abbreviation for NCA$(D, \{u, v\})$. Our verification procedure is based on the next simple observation.

[1]See Section 2.1.1 for a definition of reducibility.

Lemma 4.4 *Let v be a vertex such that $d(v) \neq r$. Then there exists a vertex u such that $d(u) = p(u) \neq r$.*

Proof: Let u be the vertex that satisfies $d(v) \rightarrow u \xrightarrow{*} v$. Then we must have $d(u) = d(v)$. Otherwise there exists a path from r to u that avoids $d(v)$, which catenated with the path $D[u, v]$ gives a path from r to v that avoids $d(v)$. □

As the previous lemma implies, in order to verify that G has no nontrivial dominators it suffices to verify that there does not exist any vertex $w \notin \mathit{succ}(r) \cup \{r\}$ that is dominated by $p(w)$. (If $w \in \mathit{succ}(r)$ then clearly $d(w) = r$.) We can do so by computing for each vertex w the maximal strongly connected component $S(G, w)$ in G that contains only descendants of w in D. Formally, let $C(G, w)$ be the set of vertices z such that (z, w) is a cycle arc entering w. By convention $w \in C(G, w)$. We define

$$S(G, w) = \{\, v \quad | \quad w \xrightarrow{*} v \text{ and } \exists\, z \in C(G, w) \text{ such that there is a path from } v \text{ to } z \text{ containing only descendants of } w \,\}. \tag{4.3}$$

Note that if $C(G, w) = \emptyset$ then $S(G, w) = \{w\}$. In order to compute and represent the $S(G, v)$ sets efficiently we define the operation *collapse*(S, v), that collapses a set $S \subseteq V$ to a vertex $v \notin S$, as follows. For each $x \in S$ and $w \notin S \cup \{v\}$, if (x, w) exists we replace it with (v, w). Similarly, if (w, x) exists we replace it with (w, v). Finally we remove S. Let $G(n) = G$ and $I(n) = S(G, n) = \{n\}$. For $k = n - 1, \ldots, 1$, we compute $I(k) = S(G(k + 1), k)$ and $G(k) =$ *collapse*$(I(k) - k, k)$. Notice that the sets $I(k) - k$ partition the set of vertices $\{i \mid 2 \leq i \leq n\}$. The sets $I(k)$ are called the *intervals* of G and can be found by computing $\nu(u, v)$ for all $(u, v) \in A$ and using disjoint set union operations [Tar76]. A simple implementation of these computations requires only a standard DSU data structure and runs in $O(m\alpha(m, n))$-time [Tar79a]. We can actually perform these computations in linear time using the linear-time DSU algorithm of [GT85], since this result also implies a linear-time version of the Aho, Hopcroft and Ullman off-line NCA algorithm [AHU76]. Note however, that the linear-time version of our algorithm is implementable only on a RAM.

```
Algorithm TVerify($G = (V, A, r)$)

    for $k = n, n-1, \ldots, 2$ do
        $label(k) \leftarrow \min\{label((j,k)) \mid j \in pred(k)\}$
        $label(k) \leftarrow \min\{label(j) \mid j \in I(k)\}$
        if $label(k) = p(k)$ and $p(k) \neq 1$ then
            return false
        endif
    done
    return true
```

Figure 4.4: Algorithm TVerify returns *false* if it finds a vertex that is dominated by its parent. It assumes that each arc $a = (u, v)$ is labeled so that $label(a) = v$ if it is a back arc, $label(a) = p(\nu(u, v))$ if it is a cross arc, and $label(a) = u$ if it is a tree or a forward arc.

The next lemma suggests how to use the intervals of G for our verification test.

Lemma 4.5 *Let k be any vertex such that $p(k) \neq 1$. Then $d(k) \neq p(k)$ if and only if there exists $j \in I(k)$ that has a predecessor in $G(k+1)$ not dominated by $p(k)$ in $G(k+1)$.*

Proof: Obviously $d(k) \neq p(k)$ if and only if there exists $j \in S(G, k)$ that has a predecessor not dominated by $p(k)$. Note that if $i \in S(G, j)$ then $i \in S(G, k)$. Thus, $S(G, k) = \bigcup_{j \in I(k)} S(G, j)$ and the Lemma follows from the definition of *collapse*(S, v). □

Figure 4.4 gives the outline of our algorithm. It assumes that we have already computed the intervals of G and that for each arc $a = (u, v)$ we have computed a label such that

$$label(a) = \begin{cases} v, & a \text{ is a back arc} \\ u, & a \text{ is a forward or a tree arc} \\ p(\nu(u,v)), & a \text{ is a cross arc} \end{cases} .$$

The algorithm processes the vertices in reverse preorder. For each vertex k it computes $label(k)$, which is the minimum of the labels of the incoming arcs of k and of the labels

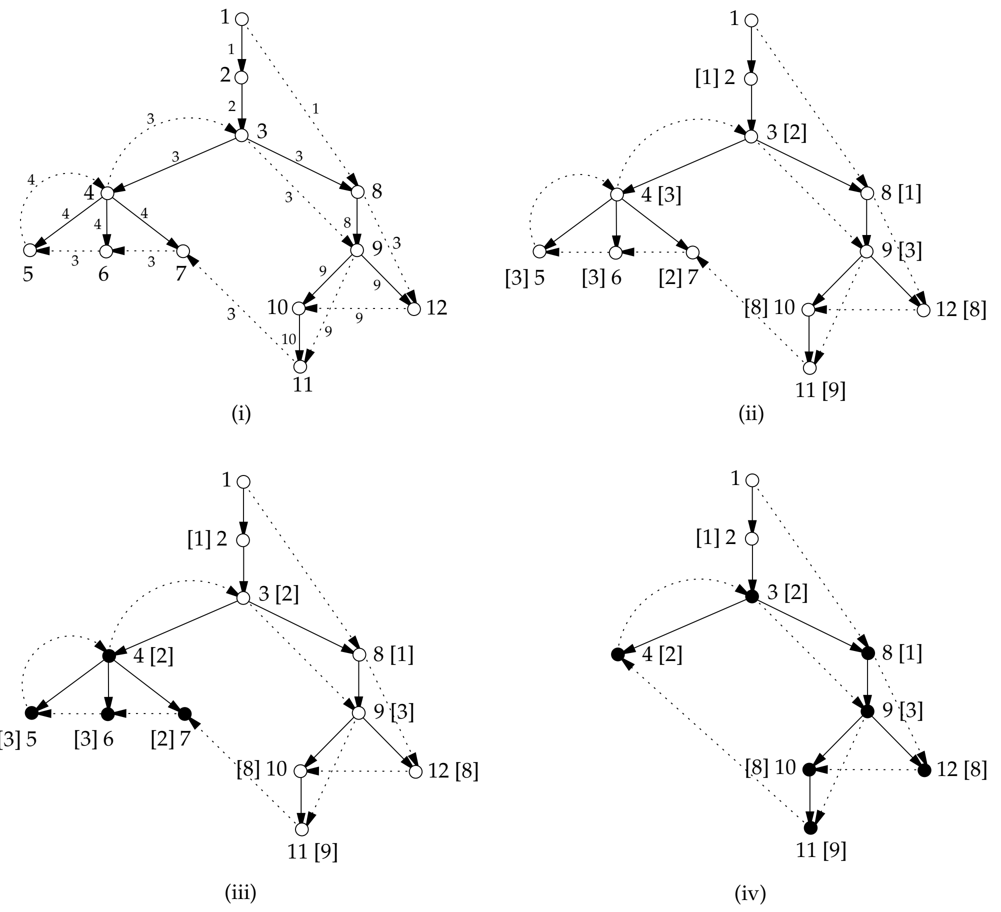


Figure 4.5: Example of the execution of TVerify. (i) The input graph with the arcs already labeled. Dotted arcs do not belong to the DFS tree. (ii) The graph after labeling each vertex by the minimum label of its incoming arcs. (We present this step as being performed separately to make the example clearer.) (iii) The situation when we process vertex 4. All the vertices with higher preorder numbers have already passed the test. We have $I(4) = \{4, 5, 6, 7\}$ (filled nodes), so the new label of 4 is $2 < p(4)$. (iv) The situation when we process vertex 3. We have collapsed $I(4) - 4$ to 4 and now $I(3) = \{3, 4, 8, 9, 10, 11, 12\}$, so the new label of 3 is $1 < p(3)$.

of the vertices in $I(k)$. It is important to note that this is equivalent to labeling vertex k in $G(k)$ by setting *label*(k) equal to the minimum label of all the arcs entering k in $G(k)$. If *label*(k) equals $p(k)$ the algorithm exits and reports that k is dominated by $p(k)$. It is clear that TVerify runs in linear time given the intervals of G and the arc labels. Figure 4.5 gives an example of the execution of this algorithm.

Lemma 4.6 *Algorithm* TVerify *is correct.*

Proof: We show by induction on k that if the algorithm does not return *false* after processing k then k is not dominated by $j = p(k)$. The basis $k = n$ is straightforward: the only arcs that may enter n, excluding the tree arc, are forward arcs. Assuming that $j \neq 1$, *label*$(n) < j$ if and only if there exists a forward arc entering n. Suppose that the algorithm has correctly verified the dominators of the vertices $n, n-1, \ldots, k+1$. By Lemma 4.5 it suffices to show that for any $z \in I(k)$ we have *label*$(z) < j$ if and only if z is not dominated by j in $G(k+1)$. Clearly if no such z exists then $I(k)$ is dominated by j and the algorithm correctly reports failure. (Remember that for each vertex $z \in I(k) - k$ there can only exist tree, forward or cross arcs entering z in $G(k+1)$.) Now assume that there is such a z. If there exists an arc (x, z) such that $\nu(z, x) < j$ then by Lemma 4.5, k is not dominated by j in G and the algorithm sets *label*$(k) < j$. Next assume that $\nu(z, x) = j$. Let l be the sibling of k that is an ancestor of x in D. Since l passed the test we already know that it is not dominated by j, and therefore z is not dominated by j. The algorithm will set *label*$(k) < j$ and k correctly passes the test. $\square$

Chapter 5

Ancestor-Dominance Spanning Trees

In this chapter we explore the relation between dominators and spanning trees. Our central result is a linear-time algorithm than constructs two spanning trees of a flowgraph G that satisfy the the following theorem:

Theorem 5.1 *Any flowgraph $G = (V, A, r)$ has two spanning trees, T_1 and T_2, that satisfy the following* ancestor-dominance *property:*

$$T_1[r, v] \cap T_2[r, v] = \mathit{dom}(v), \text{ for any } v \in V.$$

We call two spanning trees that satisfy the ancestor-dominance property *ancestor-dominance spanning trees*. In Section 5.4 we present a surprisingly simple algorithm that constructs such two spanning trees in $O(m\alpha(m, n))$-time. Furthermore, by applying the techniques of Chapter 3, we can construct a linear-time version of our algorithm. We have already seen in Section 2.1.1 that the existence of two ancestor-dominance spanning trees implies an optimal ordering of the calculations in the iterative algorithm of Allen and Cocke [AC72], so that it builds the dominator tree in one iteration. The ancestor-dominance property generalizes the notion of *independent spanning trees* that has been studied by various authors. Our work uses entirely different techniques and, in contrast to the related results on independent spanning trees, is not inductive.

5.1 Independent spanning trees

Let $\mathcal{T} = \{T_1, T_2, \ldots, T_k\}$ be a collection of spanning trees of G. These spanning trees are called *independent* if the following property holds:

> For any vertex v and any two distinct integers i and j in $[1, k]$, the paths $T_i(r, v)$ and $T_j(r, v)$ are vertex-disjoint.

Frank conjectured that any strongly k-connected graph contains k independent spanning trees, as an analog of a well-known theorem of Edmonds on edge-disjoint branchings [Edm70]. Edmonds's result states that G has k edge-disjoint branchings rooted at r if and only if, for any vertex v, there are k edge-disjoint paths from r to v. (See also [Edm72] for a different characterization of edge-disjoint branchings.) Franks's conjecture was disproved in 1995 by Huck [Huc95], who showed that for any $k \geq 3$ there is a k-connected graph that does not have k independent spanning trees. This disproof of Frank's conjecture does not hold in special cases. For instance, it does not apply to planar graphs [Huc99b]. The case $k = 2$ has proved earlier by Whitty [Whi87]. Actually Whitty proved a more general result, which we can state as follows:

Theorem 5.2 [Whi87] *Let $G = (V, A, r)$ be a flowgraph that, for any vertex v, has two vertex-disjoint paths from r to v. Then, G has two spanning trees, T_1 and T_2, such that for any pair of vertices u and v:*

$$T_1(r, v) \cap T_2(r, u) \neq \emptyset \Rightarrow T_1(r, u) \cap T_2(r, v) = \emptyset.$$

We call two spanning trees that satisfy the above theorem, *strongly independent*. Whitty gave a rather complicated but constructive proof of his theorem, and he claimed polynomial running time for the algorithm implied by his construction. The exact complexity of his construction is not specified but seems to require $\Theta(mn)$ time. Simpler constructions were given independently by Plehn [Ple91], and Cheriyan and Reif [CR92]. Both these constructions are based on an extension of the notion of *st-numberings* [LEC67, ET76] to digraphs, which Cheriyan and Reif call *directed st-numberings*. Again the exact complexity of these constructions has not been determined. Later, Huck [Huc95] presented a construction of two independent spanning trees with $O(mn)$ running time. To put Theorem 5.2 into the context of dominators, we observe that Menger's theorem [Die00] implies that G has, for any vertex v, two vertex-disjoint paths from r to v, if and only if all vertices of G have only trivial dominators.

In acyclic graphs a statement related to Frank's conjecture is true. Specifically, let G be an acyclic directed graph, k-connected to a vertex t. That means, for each $v \neq t$, there are k vertex-disjoint paths from v to t. Then G has k independent sink trees directed towards t [Huc99a]. The same result was proved independently in [ABHS00]; there, the authors present an algorithm that constructs these k trees in $O(k^2 n + km)$-time, starting from a topological order of G. A simple, linear-time construction for the case $k = 2$ was given independently by Wirth [Wir04]. His algorithm initially computes a topological order of the vertices and arranges the successor lists from deepest to shallowest successor, with respect to the topological order. Then, with a second DFS it builds T_1 and constructs the *residual* graph H by deleting the edges of T_1 that are not adjacent to the root. Finally the algorithm can pick *any* spanning tree of H to be T_2.

Itai and Rodeh proposed an analogous statement to Frank's conjecture for undirected graphs [IR84, IR88]. Specifically they conjectured that for any k-connected (undirected) graph $G = (V, E)$ and for any vertex $v \in V$, G has k independent spanning trees rooted at v. Itai and Rodeh proved their conjecture for the case $k = 2$, and gave a linear-time construction. The case $k = 3$ was proved by Cheriyan and Maheshwari [CM88], who also gave a corresponding $O(n^2)$-time algorithm, and by Itai and Zehavi [IZ89]. Recently, Curran, Lee and Yu [CLY] provided a $O(n^3)$-time algorithm that constructs four independent spanning trees of a 4-connected graph, thus proving the $k = 4$ case.

5.2 Directed st-numberings

The following result is due to Plehn [Ple91], and Cheriyan and Reif [CR92]:

Theorem 5.3 *Let $G = (V, A, r)$ be a flowgraph that, for any v, has two vertex-disjoint paths from r to v. Also, let X be the set consisting of r and the vertices that have r as their unique predecessor in G. Then, there exists a numbering*

$$\pi : V \mapsto \{1, \ldots, n\},$$

such that each vertex $v \in V \setminus X$ has predecessors u and w that satisfy

$$\pi(u) < \pi(v) < \pi(w),$$

and each vertex $v \in X - r$ satisfies

$$\pi(r) < \pi(v).$$

Given this result, Theorem 5.2 follows easily. We can construct two strongly independent spanning trees as follows: For each $v \notin X$ we pick (u, v) in T_1 and (w, v) in T_2, where u and w are the predecessors of v as stated in the theorem. Finally, for each $v \in X - r$ we can include (r, v) in both T_1 and T_2. Now consider any two vertices x and y, not necessarily distinct, and assume $\pi(x) \leq \pi(y)$. Then, $T_1(r, x)$ and $T_2(r, y)$ must be disjoint, since any $z \in T_1(r, x)$ satisfies $\pi(z) < \pi(x)$, and any $w \in T_2(r, y)$ satisfies $\pi(w) > \pi(y) \geq \pi(x)$.

Conversely, suppose that we have two strongly independent spanning trees T_1 and T_2. We say that a directed st-numbering π is *consistent* with T_1 and T_2 if the following conditions hold:

- $(x, y) \in T_1$ implies $\pi(x) < \pi(y)$.
- $(x, y) \in T_2 - r$ implies $\pi(x) > \pi(y)$.

Hence, we can think of these trees as imposing a partial order on the nodes of G, and then π is a linear extension of this partial order. Suppose now that we form a graph $\Gamma =$

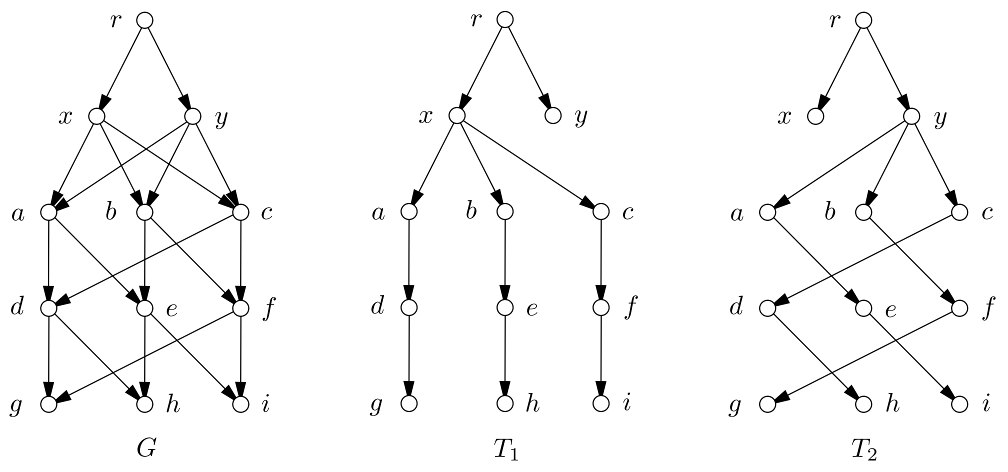


Figure 5.1: Independence does not imply strong independence. T_1 and T_2 are independent spanning trees of G, but not strongly independent. Consider, for example, the pair of vertices g and h; we have $T_1(r, g) \cap T_2(r, h) = \{d\}$ and $T_1(r, h) \cap T_2(r, g) = \{b\}$.

(V, A_Γ, r) starting from T_1 and adding the arcs in $T_2 - r$ with their orientation reversed. Then, T_1 and T_2 have a consistent directed st-numbering if and only if Γ is acyclic. Note that in this case we can construct a directed st-numbering of G by taking the topological order of Γ. Unfortunately, as we show in the next section, it is not true that any pair of strongly independent spanning trees has a consistent directed st-numbering.

5.3 Hierarchy

We give two examples which show that the properties of independence, strong independence, and directed st-numberings are distinct in the following sense. Figure 5.1 shows that two independent spanning trees are not necessarily strongly independent. Similarly, two strongly independent spanning trees do not necessarily have a consistent directed st-numbering, as illustrated by Figure 5.2.

However, as we show in the next section, these notions are computationally equivalent: Given any two independent spanning tress we can construct a directed st-numbering (and therefore two strongly independent spanning trees) in linear time.

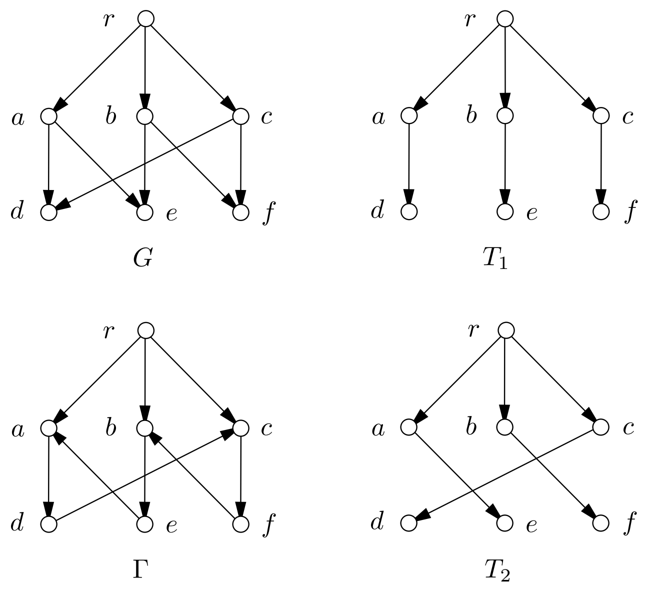


Figure 5.2: Strong independence does not imply a directed st-numbering. T_1 and T_2 are strongly independent spanning trees of G, but Γ contains the cycle (a, d, c, f, b, e, a).

5.3.1 Directed st-numberings via independent spanning trees

Here we show how to construct efficiently a directed st-numbering π of a flowgraph G, using two (precomputed) independent spanning trees, T_1 and T_2, of G. This algorithm is essentially the one given by Cheriyan and Reif, but exploits the additional information given by the independent trees in order to achieve linear running time.

We consider the graph G^n formed by the arcs in $T_1 \equiv T_1^n$ and $T_2 \equiv T_2^n$. The process we use to construct $\pi \equiv \pi^n$ runs in two phases, each consisting of $n-2$ rounds. During the i-th round of the first phase we may remove or replace some arcs in G^{n-i+1}, to form a graph G^{n-i} with $n-i$ vertices. We also perform the corresponding changes to T_1^{n-i+1} and T_2^{n-i+1}, forming T_1^{n-i} and T_2^{n-i}. We stop at G^2, for which a valid numbering π^2 is obvious; there is only one arc (r, x) so we can assign $\pi^2(r) = 1$ and $\pi^2(x) = 2$. Our goal is to maintain the following invariants:

A1. G^{n-i} has $n-i$ vertices and at most $2(n-i)-2$ arcs.

A2. G^{n-i} has at least one vertex with out-degree at most 1.

A3. T_1^{n-i} and T_2^{n-i} are independent spanning trees of G^{n-i}.

During the second phase, we perform the reverse sequence of operations constructing a numbering π^{i+1} from π^i. We will maintain the following invariant:

B1. π^i is a directed st-numbering for G^i.

Now we give the details of each phase, assuming that the invariants hold.

First phase. Consider the i-th round of this phase. We have a graph $G' = G^{n-i+1}$ and two independent spanning trees T_1^{n-i+1} and T_2^{n-i+1}. Let y be a vertex with out-degree in G', $\delta^+_{G'}(y) \leq 1$; by A2, G^{n-i+1} contains at least one such vertex. Then y must be a leaf in at least one of the two spanning trees. Assume that y is a leaf in T_1^{n-i+1}. (If y is a leaf in T_2^{n-i+1} we apply the symmetric steps.) First we remove y and its adjacent arcs (entering or leaving y). If y is also a leaf in T_2^{n-i+1} then we are done. Otherwise, let z be the child of y in T_2^{n-i+1}. Also, let x be the parent of y in T_2^{n-i+1}. We form G^{n-i} by inserting the arc (x, z). We form T_2^{n-i} by making x the parent of y. This completes the description of the i-th round.

Second phase. Consider the i-th round. Here, we have a graph G^i, two independent spanning trees T_1^i and T_2^i of G^i, and a directed st-numbering π^i for G^i. Suppose y was a leaf of T_1^{i+1} that we removed during the first phase to get T_1^i. (We apply the symmetric steps if y was a leaf in T_2^{i+1}.) Let w be the parent of y in T_1^{i+1} and x be the parent of y in T_2^{i+1}. Note that since T_1^i and T_2^i are independent, we either have $x \neq w$, or $x = w = r$. Consider the linked list L^i of the vertices in G^i that represents the numbering π^i in ascending order (starting from r). In order to get the desired numbering π^{i+1} we have to insert y in a suitable place. First suppose that y is a leaf in T_2^{i+1}. Then we get π^{i+1} by inserting y anywhere between x and w in L^i if $x \neq w$, or right after r if $x = w$. Now suppose that y has a child z in T_2^{i+1}. Let t be the parent of z in T_1^{i+1}. We consider the following cases:

(i). $w = x = t = r$; we insert y immediately after z.

(ii). $w = x = r$ and $t \neq r$ we insert y immediately before z.

(iii). $\pi^i(w) > \pi^i(x)$; we insert y immediately after x.

(iv). $\pi^i(w) < \pi^i(x)$; we insert y immediately before x.

Correctness

The next two lemmas prove the correctness of this algorithm. We start by showing that during the first phase the invariants A1-A3 are maintained. Then the correctness follows by showing that during the second phase the invariant B1 holds.

Lemma 5.4 *The first phase of the algorithm maintains the invariants A1, A2 and A3.*

Proof: We prove the Lemma by induction. For basis consider the graph G^n and its spanning trees T_1^n and T_2^n. Since T_1^n and T_2^n are independent spanning trees, A1 and A3 hold. We note that $\delta^+_{G^n}(r) \geq 2$ (otherwise G^n has non-trivial dominators) and $\delta^-_{G^n}(r) = 0$. Hence, there are at most $2n - 4$ arcs in the subgraph induced by $V - r$, so not all the vertices in this subgraph can have out-degree more than 2. This proves A2.

Now suppose that the invariants hold for G^{n-i+1}, T_1^{n-i+1} and T_2^{n-i+1}. Consider the graph G^{n-i} and the trees T_1^{n-i} and T_2^{n-i} obtained by removing a vertex y of out-degree less or equal to 1. Clearly, T_1^{n-i} and T_2^{n-i} are spanning trees of G^{n-i}. Hence, the same reasoning with the base case shows that A1 and A2 hold in G^{n-i}. Also, since the ancestors of any vertex in T_j^{n-i} are a subset of its ancestors in T_j^{n-i+1} ($j = 1, 2$), T_1^{n-i} and T_2^{n-i} are independent. □

Lemma 5.5 *The second phase of the algorithm maintains the invariant B1.*

Proof: Again we apply induction on the step number. The base case for G^2 trivially holds. For the induction step we assume that π^i is a valid numbering for G^i. For π^{i+1} we only need to consider y and z (if it exists), since these are the only vertices that change predecessors.

First we consider the case where y is a leaf in both T_1^{i+1} and T_2^{i+1}. If $w = x$ then r is the only predecessor of y in G^{i+1}, so π^{i+1} is valid if we insert y anywhere after r. If $w \neq x$, then π^{i+1} is valid after inserting y anywhere between w and x.

Now suppose that z exists and $w = x (= r)$. If $t = r$, then the algorithm inserts y immediately after z. In that case

$$L^{i+1} = (r, \ldots, z, y, \ldots),$$

and π^{i+1} is valid since r is the only predecessor of y. Otherwise ($t \neq r$), we must have $\pi^i(z) < \pi^i(t)$, so after inserting y immediately before z we get

$$L^{i+1} = (r, \ldots, y, z, \ldots, t, \ldots)$$

which again gives a valid numbering.

Next consider that $\pi^i(w) > \pi^i(x)$. Then,

$$\begin{aligned} L^i &= (r, \ldots, x, \ldots, w, \ldots, z, \ldots), \text{ or} \\ L^i &= (r, \ldots, x, \ldots, z, \ldots, w, \ldots), \text{ or} \\ L^i &= (r, \ldots, z, \ldots, x, \ldots, w, \ldots). \end{aligned}$$

In all cases, if we insert y immediately after x we get a valid numbering.

Finally, suppose that $\pi^i(w) < \pi^i(x)$. Now,

$$\begin{aligned} L^i &= (r, \ldots, w, \ldots, x, \ldots, z, \ldots), \text{ or} \\ L^i &= (r, \ldots, w, \ldots, z, \ldots, x, \ldots), \text{ or} \\ L^i &= (r, \ldots, z, \ldots, w, \ldots, x, \ldots). \end{aligned}$$

In all cases, if we insert y immediately before x we get a valid numbering. □

Running time

Given two independent trees we can construct G^n, keep in an array the out-degree of each vertex, and construct a list with the vertices with out-degree either 0 or 1 in $O(n)$ time. The algorithm maintains explicitly the current spanning trees at each round; for each vertex v it keeps two parent pointers $p_{T_1}(v)$ and $p_{T_2}(v)$ and two children lists $\mathit{chd}_{T_1}(v)$ and $\mathit{chd}_{T_2}(v)$. It is straightforward to maintain the array and the linked list in constant time per round since at most four arcs change.

During the first phase, when vertex y is removed it is inserted at the front of a linked list Λ that keeps track of the order of the operations. Consider the state of the children lists of y at that moment; together they contain at most one vertex z which can be found in constant time. The status of the corresponding pointers remains fixed for the rest of this phase. Also, the children lists of $p_{T_1}(y)$ and $p_{T_2}(y)$ must be updated. Suppose without loss of generality that $\mathit{chd}_{T_1}(v)$ is *null* and $\mathit{chd}_{T_2}(v) = (z)$. Then, we remove y from $p_{T_1}(y)$ and change the record of y to point z in $p_{T_2}(y)$. Again, all these updates can be done in constant time, since we have access to pointers for z and y.

Now consider the operations performed in the second phase. At each round we must remove the current front vertex y of Λ and insert it back to the trees. Clearly, the operations that we mentioned in the first phase can be reversed, each in constant time. Furthermore, we have to implement the list L^i that represents π^i. To that end we can use a data structure for the list order maintenance problem [DS87, BCD$^+$02]; the total time consumed by such a data structure is $O(n)$. Hence, with this implementation both phases run in $O(n)$ time.

5.4 A fast algorithm for ancestor-dominance spanning trees

In this section we present a simple, fast algorithm that constructs two ancestor-dominance spanning trees. Our algorithm uses the concept of semidominators (see Section 2.2). Therefore, our first step is to perform a DFS on the input flowgraph G. Let D be the corresponding DFS tree. For any pair of vertices u and v we let $\nu(u,v) = \mathrm{NCA}(D, \{u,v\})$.

For any $v \in V - r$, we define $t(v)$ to be a predecessor of v that belongs to an sdom path from $s(v)$ to v. Such vertices can be found easily during the computation of the semidominators. We will need the following statement about semidominators:

Lemma 5.6 *Let w be a vertex such that $s(w)$ is not a predecessor of w. Then there exists a vertex $h(w)$ such that $\nu(w, t(w)) \xrightarrow{+} h(w) \xrightarrow{*} t(w)$ and $s(h(w)) = s(w)$.*

Proof: Consider the sdom path P from $s(w)$ to w that contains $t(w)$. By the definition of the sdom path we have $x > w$ for every vertex $x \in Q = P \setminus \{w, s(w)\}$. Let x be the smallest vertex in Q. Clearly $s(x) = s(w)$. If $x = t(w)$ then $h(w) = t(w)$ and we are done. Otherwise, $x < t(w)$ so by Lemma A.1, Q must contain a vertex y which is a common ancestor of x and $t(w)$. Then we must have $y = x$ since x is the smallest vertex in Q. Therefore, $x \xrightarrow{+} t(w)$. Also since $x > w$ we have $\nu(w, t(w)) \xrightarrow{+} x$ and the Lemma holds for $h(w) = x$. □

Consider the flowgraph $G_{\min} = (V, A_{\min}, r)$, where $A_{\min}$ is the collection of arcs

$$\{(p(v), v) \mid v \in V - r\} \cup \{(t(v), v) \mid v \in V - r\}.$$

Notice that $A_{\min}$ contains two copies of $(p(v), v)$ for each v that satisfies $t(v) = p(v)$. Also every vertex has in-degree 2 except for r, which has in-degree 0. The next lemma shows that G and $G_{\min}$ have the same dominators.

Lemma 5.7 *The flowgraphs G and $G_{\min}$ have the same dominators.*

Proof: First we prove by induction that for any $v \in V$ the semidominator of v in $G_{\min}$, denoted by $s_{\min}(v)$, is $s(v)$. Clearly $s_{\min}(v) \geq s(v)$ since we deleted arcs from G. We consider the vertices in reverse preorder. For the base case we have $v = n$ and $t(v) = s(v)$, since every predecessors of v is a proper ancestor of v, so $s_{\min}(v) = s(v)$. For the induction step we consider the case $s(v) \neq t(v)$ since the result is obvious for $s(v) = t(v)$. By Lemma 5.6 we have that $\nu(v, t(v)) \xrightarrow{+} h(v) \xrightarrow{*} t(v)$, $s_{\min}(h(v)) = s_{\min}(v)$ and $s(v) = s(h(v))$. By

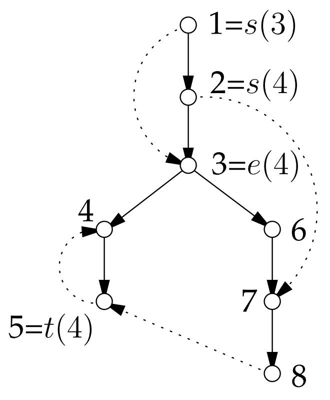


Figure 5.3: Functions s, e and t.

induction hypothesis $s_{\min}(t(v)) = s(t(v))$, thus $s_{\min}(v) = s(v)$. Then Lemma 2.5 implies $G_{\min}$ has the same dominators as G. □

The previous lemma together with the fact that any spanning tree of $G_{\min}$ is also a spanning tree of G, implies that it is sufficient to construct ancestor-dominance spanning trees for $G_{\min}$. Henceforth we will assume $G_{\min} \equiv G$.

5.4.1 Algorithm.

For any $v \neq r$, we define

$$\Sigma(v) = \{x \mid s(v) \xrightarrow{+} x \xrightarrow{*} v\}$$

and

$$E(v) = \{x \mid x \in \Sigma(v) \text{ and } s(x) \leq s(y) \text{ for all } y \in \Sigma(v)\}.$$

Also we define $e(v)$ to be the minimum vertex in $E(v)$.[1] Note that by Lemma 2.5 we have $s(e(v)) = s(v)$ if and only if $s(v) = d(v)$. Figure 5.3 illustrates the definitions used in our construction.

Figure 5.4 shows the method we use to build the two spanning trees; a blue tree B and a red tree R. We call a vertex $v \neq r$ *blue* if $(t(v), v) \in B$ and *red* otherwise. An equivalent way to state the construction is:

[1]In Section 3.4 we defined $e(v)$ to be any vertex in $E(v)$. Here we pick the minimum such vertex as it turns out that this choice simplifies our proofs.

```
Algorithm STrees($G = (V, A, r)$)

    for $k = 2, 3, \ldots, n$ do
        if $s(e(k)) = s(k)$ or $(t(e(k)), e(k)) \in R$ then
            { add $(t(k), k)$ to $B$;
              add $(p(k), k)$ to $R$ }
        else
            { add $(t(k), k)$ to $R$;
              add $(p(k), k)$ to $B$ }
        endif
    done
```

Figure 5.4: Algorithm STrees constructs two ancestor-dominance spanning trees B and R.

Color v blue if $s(e(v)) = s(v)$ or $e(v)$ is red; color v red otherwise.

Figure 5.5 gives an example of the construction. Even though this construction is simple (given the functions s, e and t), verifying its correctness is intricate. We prove first that B and R are acyclic and hence are trees, and second that corresponding paths in B and R are disjoint. Both of these steps require some preliminary ground work.

5.4.2 Properties of B and R.

We begin the analysis of our algorithm with two lemmas that relate the colors of certain vertices.

Lemma 5.8 *Let v and w be vertices such that $v \xrightarrow{*} w$, $s(v) = s(w)$, and $s(x) \geq s(v)$ for all x such that $v \xrightarrow{+} x \xrightarrow{+} w$. Then v and w are the same color.*

Proof: The hypotheses of the Lemma and the definition of the function e imply that $e(v) = e(w)$. This and $s(v) = s(w)$ imply that v and w are the same color. □

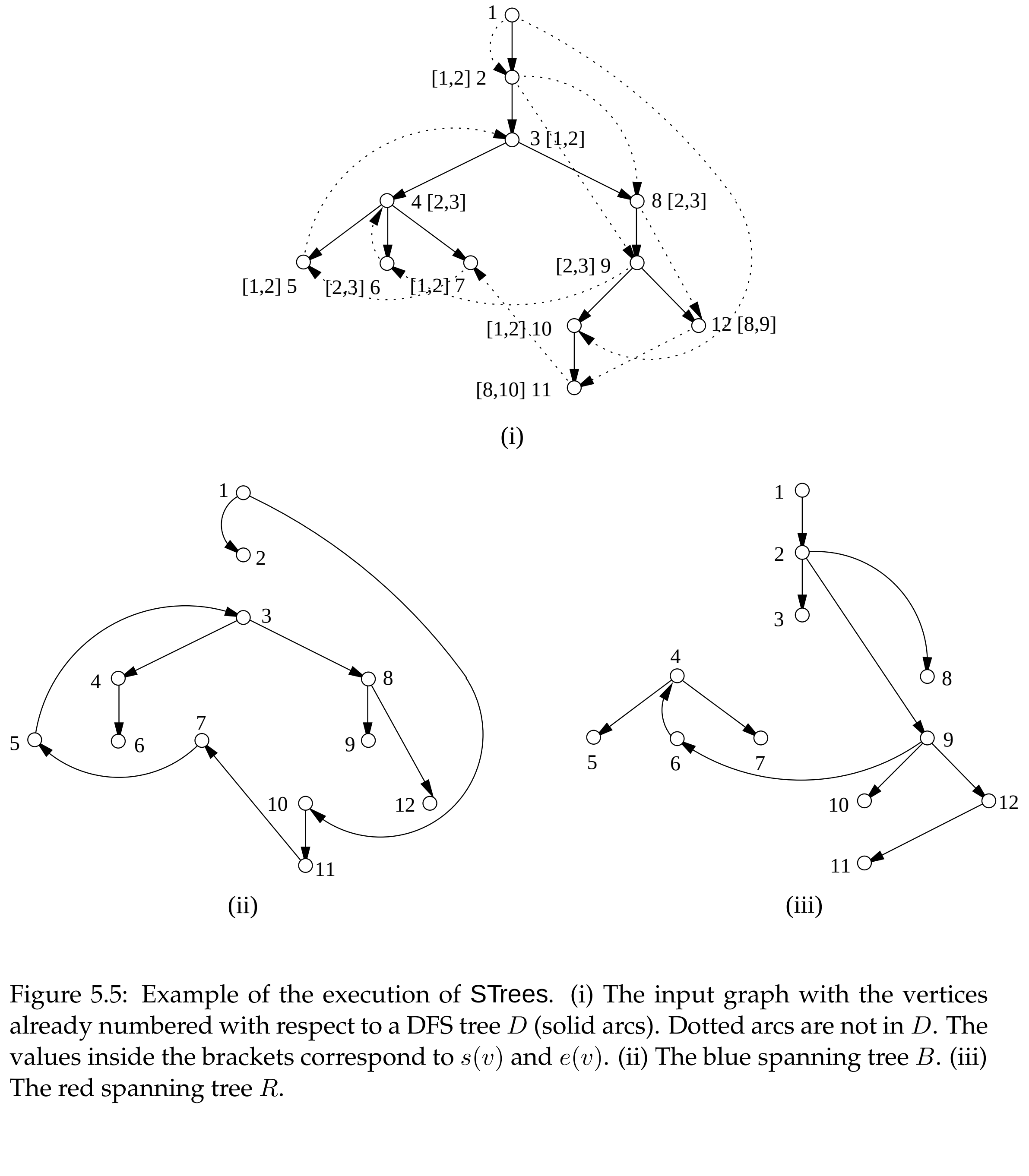


Figure 5.5: Example of the execution of STrees. (i) The input graph with the vertices already numbered with respect to a DFS tree D (solid arcs). Dotted arcs are not in D. The values inside the brackets correspond to $s(v)$ and $e(v)$. (ii) The blue spanning tree B. (iii) The red spanning tree R.

Lemma 5.8 implies that B and R remain the same if, for each $v \neq r$, we let $e(v)$ be *any* vertex $x \in E(v)$.

Lemma 5.9 *Let x, y and z be vertices that satisfy the following conditions:*

(i) $x \xrightarrow{+} y \xrightarrow{+} z$ *or* $x \xrightarrow{+} z \xrightarrow{+} y$,

(ii) $s(x) < s(y) < s(z) < x$,

(iii) $s(w) \geq s(x)$, *for all w such that* $\min\{y, z\} \xrightarrow{*} w \xrightarrow{*} \max\{y, z\}$, *and*

(iv) *x and z are the same color.*

Then y is the same color as x and z.

Proof: Suppose that the Lemma is false. Choose three vertices x, y and z that violate the Lemma and such that x is minimum. Since $x \in \Sigma(z)$, $s(e(z)) \leq s(x) < s(z)$. So $e(z)$ and z have different colors, which implies that $e(z) \neq x$. If $s(e(z)) = s(x)$ then Lemma 5.8 implies that $e(z)$ and x have the same color, a contradiction. Thus $s(e(z)) < s(x)$ and (iii) implies that $e(z) \xrightarrow{+} \min\{y, z\}$. Since y and z have different colors and $e(z) \in \Sigma(y)$, it must be the case that $e(y) \xrightarrow{*} s(z)$ and $s(e(y)) < s(e(z))$. But then $e(y), e(z)$ and x violate the Lemma, contradicting the choice of x. $\square$

Next we prove that B and R are trees.

Lemma 5.10 *Neither B nor R contains a cycle.*

Proof: We shall derive a contradiction to the assumption that B contains a cycle; the same argument applies to R. Given a cycle in B, let v be the minimum vertex on the cycle. Then $v \neq r$ (since r contains no incoming arcs), and $s(v) < v$. Also, by the properties of DFS (Lemma A.1), $(t(v), v)$ is a cycle arc, v is blue and all vertices on the cycle are descendants of v. Let w be the first vertex after v on the cycle such that w is blue and $s(w) < v$. (If v is the only such vertex on the cycle, then $w = v$.) Then $v \xrightarrow{+} t(w)$, since $v = t(w)$ would imply $s(w) = t(w) = v$, which contradicts $s(w) < v$. By Lemma 5.6, $u = h(w)$ satisfies

$v \xrightarrow{+} u \xrightarrow{*} t(w)$ and $s(u) = s(w) < v$. We claim that u is blue. Indeed, v is a candidate for both $e(u)$ and $e(w)$, which means that $\max\{s(e(u)), s(e(w))\} \leq s(v)$. Also, the definition of $s(v)$ implies that $s(x) \geq s(v)$ for any vertex x that is a descendant of v and an ancestor of either u or w. It follows that $e(u) = e(w)$ is an ancestor of v, and hence u is the same color as w; namely, blue. Let z be the vertex on the cycle such that all vertices on the cycle from z to $t(w)$ (inclusive) are descendants of u, but the predecessor y of z on the cycle is not a descendant of u. There must be such a vertex z, since v is on the cycle but not a descendant of u. Furthermore, starting from v, z precedes w on the cycle. It cannot be the case that $z = u$, for then u would be on the cycle after v but before w, contradicting the choice of w. Thus $u \xrightarrow{+} z$, z is blue, and $t(z) = y$ is not a descendant of u. But $t(z)$ not a descendant of u implies $s(z) < u$. It cannot be the case that $s(z) < v$, for this would contradict the choice of w. Thus $v \xrightarrow{*} s(z) \xrightarrow{+} u \xrightarrow{+} z$. Then $s(e(z)) \leq s(u) < v < z$ and $e(z)$ is red. Therefore, by Lemma 5.8 and the fact $e(z) \in S(v)$, we have $s(v) < s(e(z)) < s(u)$. By Lemma 5.9, $e(z)$ must be blue, contradicting the fact that $e(z)$ is red. $\square$

To prove disjointness of paths in B and R, we need one technical lemma in addition to Lemmas 5.8 and 5.9. This lemma requires some more definitions. For each vertex $v \neq r$, define $\widehat{s}(v)$ as follows: if v is blue (red) $\widehat{s}(v)$ is the nearest ancestor x of v in B (R) such that $x < v$. By Lemma 5.10, B and R are trees rooted at r, which implies that $\widehat{s}(v)$ is well-defined. By Lemma A.1 and the definition of function s, $s(v) \xrightarrow{*} \widehat{s}(v) \xrightarrow{+} v$. Let

$$\widehat{\Sigma}(v) = \{x \mid \widehat{s}(v) \xrightarrow{+} x \xrightarrow{*} v\}$$

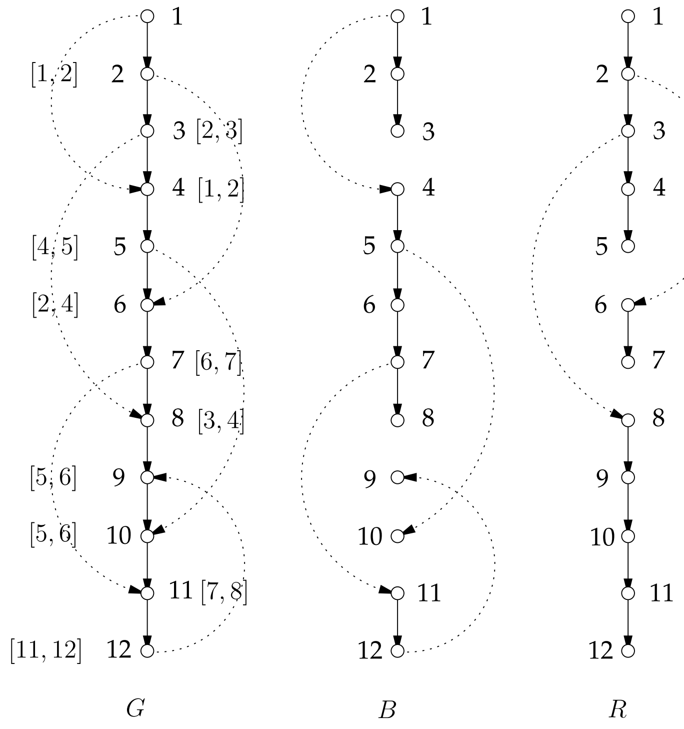
and

$$\widehat{E}(v) = \{x \mid x \in \widehat{\Sigma}(v) \text{ and } s(x) \leq s(y) \text{ for all } y \in \widehat{\Sigma}(v)\}.$$

Let $\widehat{e}(v)$ be the minimum vertex in $\widehat{E}(v)$. These definitions imply the following relation:

$$s(v) \xrightarrow{*} e(v) \xrightarrow{*} \hat{s}(v) \xrightarrow{*} \hat{e}(v) \xrightarrow{+} v.$$

In fact, all these vertices can be distinct. See Figure 5.6.



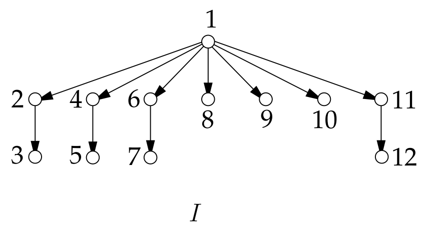


Figure 5.6: A situation where $s(v) \xrightarrow{+} e(v) \xrightarrow{+} \hat{s}(v) \xrightarrow{+} \hat{e}(v)$. The above relation holds for $v = 9$ which is blue; we have $s(9) = 5$, $e(9) = 6$, $\hat{s}(9) = 7$ and $\hat{e}(9) = 8$. The values inside the brackets correspond to $s(v)$ and $e(v)$. Duplicate arcs are not shown in $G \equiv G_{\min}$.

Lemma 5.11 *For any vertex $v \neq r$, either $\widehat{s}(v) = s(v)$, or $s(\widehat{e}(v)) < s(v)$ and $\widehat{e}(v)$ and v are different colors.*

Proof: The proof is by induction on v in decreasing order. If $\widehat{s}(v) = s(v)$ then the Lemma holds for v. Thus suppose $s(v) < \widehat{s}(v)$. If $(t(v), v)$ is a forward arc, $\widehat{s}(v) = t(v) = s(v)$, a contradiction. Thus $(t(v), v)$ is a cycle arc or a cross arc. Let $\nu = \nu(t(v), v)$ and $w = h(v)$. By Lemma 5.6, $\nu \xrightarrow{+} w \xrightarrow{*} t(v)$ and $s(w) = s(v)$. Lemma A.1 implies that $\widehat{s}(v) \xrightarrow{*} \nu$. If $\nu \xrightarrow{+} e(v)$ and $s(e(v)) < s(v)$, then $\widehat{e}(v) = e(v)$ and the Lemma holds for v by the construction of B and R. Thus suppose $e(v) \xrightarrow{*} \nu$ or $s(e(v)) = s(v)$. We claim that in either case v and w are the same color. The definition of $s(v)$ implies that $s(x) \geq s(v)$ for all x such that $\nu \xrightarrow{+} x \xrightarrow{*} w$. If $e(v) \xrightarrow{*} \nu$, then $e(v) = e(w)$ and v and w are the same color. If $s(e(v)) = s(v)$, then $s(x) \geq s(v)$ for all x such that $s(v) \xrightarrow{+} x \xrightarrow{*} \nu$, which means that $s(e(w)) = s(v) = s(w)$, and again v and w are the same color.

Suppose v and w are both blue; the symmetric argument applies if they are both red. Let z be the nearest ancestor of $t(v)$ in B such that the parent y of z in B is not a descendant of w. Vertex z is blue, since w is blue and if $z \neq w$ the blue arc (y, z) entering z cannot be a tree arc. Also $z > v$ (follows from Lemma A.1), so the Lemma holds for z by the induction hypothesis. The definition of $s(v)$ implies that $s(z) \geq s(v)$, because z is on $B(\widehat{s}(v), v)$. If $s(z) = s(v)$ and $\widehat{s}(z) = s(z)$, then $\widehat{s}(v) = \widehat{s}(z) = s(v)$, and the Lemma is true. Thus suppose $s(z) > s(v)$ or $\widehat{s}(z) > s(z)$. We claim that $s(\widehat{e}(z)) < s(v)$ and $\widehat{e}(z)$ is red. If $s(z) = s(v)$ and $\widehat{s}(z) > s(z)$, the claim follows since the Lemma holds for z. Suppose $s(z) > s(v)$. Then $z \neq w$ and $\widehat{s}(v) \xrightarrow{+} w \xrightarrow{+} z$ by the existence of the arc (y, z). Thus

$$s(e(z)) \leq s(\widehat{e}(z)) \leq s(w) = s(v) < s(z),$$

and by the construction of B and R, $e(z)$ is red. If $\widehat{s}(z) > s(z)$, $\widehat{e}(z)$ is red since the Lemma holds for z. Also $s(\widehat{e}(z)) < s(v)$, since $s(\widehat{e}(z)) = s(v) = s(w)$ implies $\widehat{e}(z)$ is blue by Lemma 5.8. Since $s(\widehat{e}(z)) < s(v)$, $\widehat{e}(z) \xrightarrow{*} \nu$, which implies $\widehat{s}(z) \xrightarrow{*} \nu$ and $\widehat{s}(v) = \widehat{s}(z)$. Either $\widehat{e}(v) = \widehat{e}(z)$, in which case the Lemma holds for v, or $\nu \xrightarrow{+} \widehat{e}(v)$ and $s(\widehat{e}(v)) < s(\widehat{e}(z))$. In this case if $s(e(v)) = s(\widehat{e}(v))$ then $\widehat{e}(v)$ is red by Lemma 5.8 and the Lemma holds for v; if $s(e(v)) < s(\widehat{e}(v))$ then $\widehat{e}(v)$ is red by Lemma 5.9 applied to $e(v)$, $\widehat{e}(v)$, and $\widehat{e}(z)$, and the Lemma holds for v. □

5.4.3 Vertex-disjointness.

Now we are ready to prove that B and R satisfy the ancestor-dominance property. First we argue that it suffices to prove that for each v the paths $B(d(v), v)$ and $R(d(v), v)$ contain no common vertex.

Lemma 5.12 *Let B and R be to spanning trees of G. Then B and R satisfy the ancestor-dominance property if and only if for any $v \neq r$, $B(d(v), v)$ and $R(d(v), v)$ are vertex-disjoint.*

Proof: The first direction is obvious; if B and R are ancestor-dominance spanning trees, then $B(d(v), v)$ and $R(d(v), v)$ must be disjoint. For the contraposition, let $d = d(v)$ and let $dom(v) = \{d_1 = r, d_2, \ldots, d_{k-1} = d, d_k = v\}$, where $d_i = d(d_{i+1})$ for $1 \leq i \leq k-1$. Then, d_i is an ancestor of d_{i+1} in both B and R, so the dominators of v appear in the same order in $B(r, v)$ and $R(r, v)$. Suppose now that $B(r, v)$ and $R(r, v)$ intersect at a vertex x such that d_B is the closest dominator of v in B that is an ancestor of x, and similarly d_R is the closest dominator of v in R that is an ancestor of x. Without loss of generality assume that d_B is an ancestor of d_R (in both B and R). If $d_B \neq d_R$ then the path $B(r, x)$ followed by $R(x, v)$ avoids d_R which is a contradiction. Hence $d_B = d_R = d_i$, and the paths $B(d_i, d_{i+1})$ and $R(d_i, d_{i+1})$ intersect at x. □

Finally the next lemma proves the vertex-disjointness of $B(d(v), v)$ and $R(d(v), v)$.

Lemma 5.13 *Let v be any vertex other than r, and let $d = d(v)$. Then the paths $B(d, v)$ and $R(d, v)$ contain no common vertex.*

Proof: Suppose to the contrary that $B(d, v)$ and $R(d, v)$ both contain a vertex $w \notin \{d, v\}$. Let x_B and x_R be the minimum vertices on $B[w, v]$ and $R[w, v]$ respectively. Neither $B[w, v]$ nor $R[w, v]$ contains d, since $B[d, v]$ and $R[d, v]$ are simple paths. In particular $d \notin \{x_B, x_R\}$. Assume $x_B \leq x_R$; the symmetric argument applies if $x_R \leq x_B$. We have that $d \xrightarrow{+} x_B$ and, by DFS (Lemma A.1), $x_B \xrightarrow{*} x_R \xrightarrow{*} v$. If $x_B \neq w$ then x_B is blue since then it is entered by a blue nontree arc. Similarly if $x_R \neq w$ then x_R is red. We have $x_B \xrightarrow{+} v$, since $x_B = v$ implies $x_R = v$ and v is both blue and red since $w \neq v$, a contradiction.

Let u be a vertex of minimum $s(u)$ such that $x_B \xrightarrow{+} u \xrightarrow{*} v$. Since x_B does not dominate v, Lemma 2.5 implies $s(u) \xrightarrow{+} x_B$. By Lemma 5.11, either $\widehat{s}(u) = s(u)$, in which case $\widehat{s}(u) \xrightarrow{+} x_B$, or $s(\widehat{e}(u)) < s(u)$, which implies by the choice of u that $\widehat{e}(u) \xrightarrow{*} x_B$, and again $\widehat{s}(u) \xrightarrow{+} x_B$. We claim that u is red. Suppose to the contrary that u is blue. Then u cannot be on $B[x_B, v]$; if it were, $\widehat{s}(u)$ would be on $B[x_B, v]$, since $x_B \xrightarrow{+} u$; but every vertex on $B[x_B, v]$ is no less than x_B, contradicting $\widehat{s}(u) \xrightarrow{+} x_B$. Let x be the first vertex on $B[x_B, v]$ such that $u \xrightarrow{+} x \xrightarrow{*} v$. Then x is blue and $x_B \xrightarrow{*} \widehat{s}(x) \xrightarrow{+} u \xrightarrow{+} x \xrightarrow{*} v$. The definition of u implies $\widehat{e}(x) = u$. If $\widehat{s}(x) = s(x)$, then $e(x) = u$, and u is red by the construction of B and R since x is blue. If $\widehat{s}(x) > s(x)$, then $u = \widehat{e}(x)$ is red by Lemma 5.11 since x is blue.

Since u is red, $u \xrightarrow{*} x_R$, because if $x_R \xrightarrow{+} u$ an argument symmetric to that in the previous paragraph shows that u is blue. Since $x_B \xrightarrow{+} u \xrightarrow{*} x_R$, $x_B \neq w$, which implies that x_B is blue.

We claim that $s(u) \leq \widehat{s}(x_B)$. Vertex w is on $B[\widehat{s}(x_B), x_B]$. Consider the path $B[\widehat{s}(x_B), w]$ followed by $R[w, v]$. This path avoids x_B. Let y be the first vertex along this path such that $x_B \xrightarrow{+} y \xrightarrow{*} v$. The part of the path from $\widehat{s}(x_B)$ to y is an sdom path for y. Hence $s(y) \leq \widehat{s}(x_B)$. By the choice of u, $s(u) \leq s(y) \leq \widehat{s}(x_B)$.

Next we claim that $s(u) < s(x_B)$. If $s(u) = s(x_B)$, u and x_B are the same color by Lemma 5.8, a contradiction. If $s(u) > s(x_B)$, then since $s(u) \leq \widehat{s}(x_B)$, Lemma 5.11 gives $s(\widehat{e}(x_B)) < s(x_B)$ and $\widehat{e}(x_B)$ is red. But then $\widehat{e}(x_B), x_B$ and u violate Lemma 5.9.

Now we claim that w is not a descendant of u. Suppose to the contrary that $u \xrightarrow{*} w$. Then the path from $s(u)$ to x_B consisting of the sdom path from $s(u)$ to u, followed by the tree path to w, followed by $B[w, x_B]$ is an sdom path for x_B, giving $s(u) \geq s(x_B)$, a contradiction.

Since w is not a descendant of u, $w \neq x_R$. Hence x_R is red. Furthermore it cannot be the case that $u \xrightarrow{*} \widehat{s}(x_R)$, since w is on $R[\widehat{s}(x_R), x_R]$, which implies $\widehat{s}(x_R) \xrightarrow{*} w$. Thus $\widehat{s}(x_R) \xrightarrow{+} u$, and $s(\widehat{e}(x_R)) \leq s(u)$. Also the path $R[\widehat{s}(x_R), w]$ followed by $B[w, x_B]$ is an sdom path for x_B, which implies $s(x_B) \leq \widehat{s}(x_R)$. If $s(x_R) = \widehat{s}(x_R)$ then $s(x_B) \leq s(x_R)$, and since $s(u) < s(x_B)$, we have $s(e(x_R)) \leq s(u) < s(x_R)$. So, by the construction of B and R, $e(x_R)$ is blue and Lemma 5.8 gives $s(e(x_R)) < s(u)$, which implies $e(x_R) \xrightarrow{+} x_B$. But then Lemma 5.9 for $e(x_R), u$ and x_B implies that u is blue, a contradiction. Hence $s(x_R) < \widehat{s}(x_R)$. Then $\widehat{e}(x_R)$ is blue by Lemma 5.11, and Lemma 5.8 implies $s(\widehat{e}(x_R)) < s(u)$. But then Lemma 5.9 for $\widehat{e}(x_R), u$ and x_B implies that u is blue, again a contradiction. □

5.4.4 Running time.

Clearly, given the s, e and t functions, algorithm STrees runs in $O(n)$ time. We can use the Lengauer-Tarjan algorithm to compute all $s(v)$, $e(v)$ and $t(v)$ in $O(m\alpha(m,n))$-time. Also, in Section 3.4 we showed that s and e can be computed in linear time. Computing t is not immediate because our algorithm modifies the input graph by inserting arcs entering a line ℓ (when the semidominators of ℓ are computed), or by inserting arcs entering a non-trivial microtree T (during the preprocessing phase that runs *microLT*). Still it is not hard to keep for each additional arc a pointer to a corresponding arc of the original graph. Hence, we can get an overall linear-time algorithm. Finally we note that the result of Alstrup et al. [AHLT99] also provides a linear-time computation of the required functions, but with significantly more complicated techniques.

5.5 Generalizations

Here we provide some generalizations of the notions of strongly independent spanning trees and directed st-numberings to flowgraphs that may have non-trivial dominators. We shall use the idea of the derived graph (see Section 4.2) to achieve these results.

Lemma 5.14 *Let $G = (V, A, r)$ be a flowgraph, and let $X \subseteq V$ be the set consisting of r and the vertices that have their immediate dominator as their unique predecessor in G. Then, there exists a numbering*

$$\pi : V \mapsto \{1, \ldots, n\},$$

such that each vertex $v \in V \setminus X$ has predecessors u and w that satisfy

$$\pi(u) < \pi(v) < \pi(w),$$

and each vertex $v \in X - r$ satisfies

$$\pi(d(v)) < \pi(v).$$

Proof: Let I be the dominator tree of G, and let d be its depth. Let G_I be the derived graph corresponding to I. Each subgraph $G_I(v)$ satisfies the premise of 5.3, and therefore has a directed st-numbering π_v. Let L_v be the list of nodes in $G_I(v)$ that represents π_v. We construct a total list L representing the desired numbering π by following a procedure that consists of d rounds. The procedure starts from r, where we assign $L \leftarrow L_r$. At the i-th round we process the vertices of G that appear at the i-th level of I; for each such vertex v we substitute the occurrence of v in L by L_v. An equivalent way to get the same numbering is as follows. For each vertex v we order its list of children in I, $\mathit{chd}_I(v)$, according to L_v. Then we perform a preorder walk on I and assign $\pi(v)$ to be the preorder number of v.

Now we argue that L satisfies the Lemma. Let v be any vertex in $V \setminus X$. We have to show that v has a predecessor u to its left in L, and a predecessor w to its right. Consider $L_{d(v)}$; since $v \notin \mathit{succ}(d(v))$, v has a predecessor u' in $G_I(d(v))$ that is located to its left in $L_{d(v)}$, and a predecessor w' in $G_I(d(v))$ to its right. Then it follows from the construction of G_I that $u' \in I(r, u]$ and $w' \in I(r, w]$. Finally, we note that all the descendants of u' in I are located to the left of v in L, and similarly all the descendants of w' in I are located to the right of v in L. $\square$

Using the previous fact we can show the following extension of strongly independent spanning trees.

Lemma 5.15 *Let $G = (V, A, r)$ be a flowgraph. Then, G has two spanning trees, T_1 and T_2, such that for any pair of vertices u and v:*

$$T_i[r, v] \cap T_j[r, u] = dom(v) \cap dom(u),$$

for $i \neq j$ in $\{1, 2\}$.

Proof: Let $X \subseteq V$ be the set consisting of r and the vertices that have their immediate dominator as their unique predecessor in G. Let π be a numbering satisfying Lemma 5.14. We construct T_1 and T_2 as in Section 5.2: For each $v \notin X$ we insert (u, v) in T_1 and (w, v) in T_2, where u and w are in *pred*(v) and satisfy $\pi(u) < \pi(v) < \pi(w)$. Finally, for each $v \in X - r$, we include $(d(v), v)$ to both trees.

Now we show that this construction satisfies the Lemma. First we establish that for any vertex v,

$$T_1[r, v] \cap T_2[r, v] = dom(v).$$

By Lemma 5.12 it suffices to show that $T_1(d(v), v)$ and $T_2(d(v), v)$ are vertex-disjoint, which follows immediately from the definition of π and the construction of T_1 and T_2.

Now we consider any pair of vertices x and y such that $\pi(x) \leq \pi(y)$. Let $d = \text{NCA}(I, \{x, y\})$, i.e., the nearest common dominator of x and y. It remains to argue that the paths $T_1(d, x)$ and $T_2(d, y)$ are vertex-disjoint. This is clear when $x = y$, since then $d = x$. So suppose $x \neq y$. Let d_x be the child of d in I that is an ancestor of x in I. Define d_y analogously. By the definition of d, we have $d_x \neq d_y$. Remember from the proof of Lemma 5.14 that π is obtained from a preoder numbering of I, hence we have:

(a) For any descendant w of d_x and any descendant z of d_y in I, $\pi(w) < \pi(z)$.

We also make the following observations:

(b) $T_1[d, x] = T_1[d, d_x] \cdot T_1[d_x, x]$ and $T_2[d, y] = T_2[d, d_y] \cdot T_2[d_y, y]$.

(c) $T_1[d_x, x]$ contains only vertices dominated by d_x, and $T_2[d_y, y]$ contains only vertices dominated by d_y.

(d) For any $w \in T(d, d_x]$, $\pi(w) \leq \pi(d_x)$. Similarly, for any $z \in T(d, d_y]$, $\pi(z) \geq \pi(d_y)$.

From (a) and (c) it follows that $T_1[d_x, x]$ and $T_2[d_y, y]$ are vertex-disjoint. Also, by (a), (c) and (d) we have $T_1(d, d_x] \cap T_2[d_y, y] = \emptyset$ and $T_1[d_x, x] \cap T_2(d, d_y] = \emptyset$. Finally, (d) implies that $T_1(d, d_x]$ and $T_2(d, d_y]$ are vertex-disjoint. Therefore, from (b) we have that $T_1(d, x)$ and $T_2(d, y)$ are vertex-disjoint. $\square$

Using similar arguments we can show that Theorem 5.2 implies 5.15. Again we begin by constructing the derived graph G_I. Each subgraph $G_I(v)$ satisfies the premise of Theorem 5.2. Hence, $G_I(v)$ has two strongly independent spanning trees $T_I^1(v)$ and $T_I^2(v)$. Clearly, the collection $\bigcup_{v \in V} T_i(v)$ forms a spanning tree T_I^i of I $(i \in \{1, 2\})$. It is also easy to verify that T_I^1 and T_I^2 are strongly independent spanning trees of G_I. Now we form T_1 and T_2 from T_I^1 and T_I^2 respectively, by substituting each arc (x, y) in T_I^1 or T_I^2 with a corresponding arc (u, v) in G, so that $\gamma(u, v) = (x, y)$. Arguments analogous to the ones we used in the proof of Lemma 5.15 show that T_1 and T_2 satisfy the strong ancestor-dominance property.

Similarly, if we start from two independent spanning trees of each $G_I(v)$ then the same construction gives two ancestor-dominance spanning trees T_1 and T_2.

Chapter 6

Concluding Remarks and Open Problems

In this dissertation we have presented linear-time algorithms for finding dominators, verifying a dominator tree, and constructing independent spanning trees and related structures. Still, several relevant problems remain open. Perhaps the most challenging one is to devise a truly simple linear-time dominators algorithm, at least for the random-access model of computation. The ideas we used in Chapters 4 and 5 may lead to this result.

A related open problem is to design a simple linear-time pointer-machine verification algorithm. The linear-time version of algorithm TVerify of Section 4.5.1 requires a linear-time algorithm for a special case of the disjoint set union problem. To that end, we applied the linear-time DSU algorithm of Gabow and Tarjan [GT85]. This algorithm is implementable only on a RAM, and it solves a problem that is more general than the DSU instance that occurs within the verification algorithm. Specifically, in the DSU problem considered in [GT85] the *union* operations are restricted to always unite the set of a node v with the set of $p_T(v)$, where T is a fixed tree given *offline* (this tree represents the structure of the *union*s). However, the actual *union* and *find* operations are performed *online*. La Poutré [Pou96] showed an $\Omega(n + m\alpha(m, n))$ lower bound for DSU on pointer machines, which is still valid when the union tree T is known ahead of time. (The same lower bound was obtained previously in [Tar79b] for $m \geq n$ and was extended in [TvL84] for all m and n. Both of these results apply to pointer machines that satisfy a certain technical condition which is lifted in [Pou96]. For a survey of the results on the disjoint set union problem and its variants see [GI91].) In the verification algorithm we use DSU operations in order to compute the intervals of the flowgraph G. The structure of the *union*s is defined by a DFS tree D of G, but we also have additional information about the actual operations that will be performed. Specifically, each series of *union* operations is triggered after processing a back or cross arc of D. Furthermore, the arcs are processed in reverse postorder with respect to their destination vertex. It is not clear whether these properties can be exploited to get a linear-time verification algorithm implementable on a pointer machine.

In Chapter 5 we saw how to construct strongly independent spanning trees and directed st-numberings (as well as some generalizations) in linear time, given two ancestor-dominance trees. We conjecture, however, that the spanning trees B and R produced by algorithm STrees actually satisfy the strong ancestor-dominance property. Furthermore, if G has trivial dominators only, we conjecture that B and R have a consistent st-numbering. A related question is whether a simple linear-time construction of a directed st-numbering exists. This will allow a simple construction of an optimal processing order for the iterative algorithm of Section 2.1, which may have practical value as well. It may also lead to a simpler linear-time dominators algorithm. Furthermore, it is possible that properties similar to ancestor-dominance can be applied to speed up more general data flow problems. Such results would find wide application in the field of optimizing compilation.

Another research direction involves the analysis of path-constrained graphs. Several practical applications can be modeled as directed graphs that exhibit *path constraints*. That is, some paths through the graph may be considered *invalid*, depending on some path-wide condition. For example, a path in an inter-procedural control-flow graph is only valid if the function call and return edges it contains satisfy appropriate nesting conditions [SP81, RHS95]. Augmenting data-flow analysis algorithms with path constraint awareness is a nontrivial task, which usually results in increased algorithm complexity. In particular, there is no obvious way to incorporate path constraints into the linear-time and almost-linear-time dominators algorithms. We note that the graphs involved in inter-procedural control-flow analysis may be orders of magnitude bigger that the graphs in standard (intra-procedural) analysis. Therefore, an experimental study of algorithms for inter-procedural analysis would also be valuable.

Appendix A

Preliminaries

Here we review some basic definitions from graph theory and establish notation. For a comprehensive treatment of this material we refer to [Die00]. Also, we define the underlying models of computation and the input formats of the data for the problems we consider, that are used in order to determine the computational complexity of our algorithms.

In the main part of the dissertation we use several basic notions and results in algorithms that are not mentioned here. These topics are covered in standard books on algorithms and data structures [AHD74, Tar83, CLR91].

A.1 Directed graphs and branchings

Let $G = (V, A)$ be a directed graph (digraph). For any vertex $v \in V$, we define the *predecessors* of v as

$$\mathit{pred}(v) = \{u \mid (u, v) \in A\}.$$

Similarly, the *successors* of v are

$$\mathit{succ}(v) = \{u \mid (v, u) \in A\}.$$

We will allow G to contain multiple arcs. In that case we will assume that *pred*(v) and *succ*(v) are collections of vertices (rather than sets). The in-degree of v, denoted by $\delta^-_G(v)$, is the number of arcs entering v, i.e., $\delta^-_G(v) = |\textit{pred}(v)|$; the out-degree of v, denoted by $\delta^+_G(v)$, is the number of arcs leaving v, i.e., $\delta^+_G(v) = |\textit{succ}(v)|$. Let $P = (v_1, \dots, v_k)$ and $Q = (u_1, \dots, u_l)$ be two paths in G, such that $v_k = u_1$. Then $P \cdot Q$ denotes the path formed by catenating P with Q.

A *(spanning-out) tree*[1] rooted at r (also called an *arborescence*), is a flowgraph $T = (V, A, r)$ with root r, such that $\delta^-_T(v) = 1$ for all $v \in V - r$. For each vertex $v \in V$ there is a unique simple path from r to v. A *forest* F is a collection of trees. Below we introduce some notation for concepts defined on trees. We will use similar notation when dealing with forests.

Let T be a tree rooted at r. The *parent* of v, denoted by $p_T(v)$, is the unique predecessor of v in T. Sometimes we use the notation "$v \rightarrow_T w$" to refer to the fact that $v = p_T(w)$. The *children* of v are the successors of v in T. We will use the notation *chd*$_T(v)$ to refer to an ordered list of the children of v in T. If the unique path from r to w in T includes v, then we say that v is an *ancestor* of w, and that w is a descendant of v. We denote this relation by "$v \overset{*}{\rightarrow}_T w$"; if $v \neq w$ then v is a *proper ancestor* of w (and w is a *proper descendant* of v) and we use the notation "$v \overset{+}{\rightarrow}_T u$". If v is neither an ancestor nor a descendant of w, then we call v and w *unrelated*.

For any vertex $v \in T$, T_v denotes the subtree of T rooted at v.

For any two vertices v and w, such that v is an ancestor of w, we denote by $T[v, w]$ the path from v to w in T. Also, we use the notation $T(v, w]$ for the subpath of $T[v, w]$ that excludes v, $T[v, w)$ for the subpath of $T[v, w]$ that excludes w, and $T(v, w)$ for the subpath of $T[v, w]$ that excludes both v and w. We will apply similar notation to simple paths of a graph. For instance, if $P = (v_1, v_2, \dots, v_k)$ is a simple path, we denote the subpath of P from v_i to v_j by $P[v_i, v_j]$.

The *nearest common ancestor* (NCA) of any two vertices v and w is defined as the vertex

[1] All the trees that we consider here are rooted and directed.

in $T[r,v] \cap T[r,w]$ farthest from the root r. This concept generalizes in a straightforward way for a set of vertices; for any subset $U \subseteq V$, we let NCA(T,U) denote the nearest common ancestor of U in T, i.e., the vertex in $\bigcup_{u \in U} T[r,u]$ farthest from the root r.

A.1.1 Branchings and spanning trees

Let $R = \{r_1, \ldots, r_k\} \subseteq V$. An *R-branching* $\mathcal{B}$ of a digraph $G = (V, A)$ is a spanning forest of G, i.e., a collection of subgraphs $B_i = (V_i, A_i, r_i)$ of G, where $\bigcup_i V_i = V$ and for each vertex $v \in V \setminus R$, $\delta^{-}_{\mathcal{B}}(v) = 1$ (thus each B_i is a tree rooted at r_i). When R consists of a single vertex r we refer to $\mathcal{B}$ as an *r-branching*. Furthermore, if $\mathcal{B} = \{B\}$, then B is a *spanning tree* of G.

A.1.2 Tree traversal

Consider a tree T. A *traversal* of T visits the vertices of the tree exactly once. We will consider two systematic orders of traversing a tree: *preorder* and *postorder*. In preorder we visit each vertex before its children, and vice versa in postorder. We will assume that the children of any vertex $v \in T$ are visited according to the order they appear in *chd*$_T(v)$. A preorder (postorder) numbering for T numbers each vertex according to the order it was visited by the preorder (postorder) traversal. (See Figure A.1(ii).)

A.1.3 Graph search

Let $G = (V, A, r)$ be a flowgraph. A *search* of G starting from r follows the arcs in A until all of them are explored. The search maintains a set of visited vertices S and a set of unexplored arcs U. Initially, $S = \{r\}$ and $U = A$. We say that a vertex v becomes *inactive* after all the arcs leaving v are explored. At each step the search explores one arc (v,w) in U, such that $v \in S$. Then w becomes a visited vertex if it was not in S already. The search also produces a numbering of the vertices, by letting the i-th visited vertex have number i, and a spanning tree T of G, by assigning $p_T(w) \leftarrow v$ if w was unvisited before (v,w) was explored. Sometimes we refer to the vertices by their search numbers; for instance we can compare the numbers assigned to two vertices and write $v < w$, which means that the search visited v before w.

We will mostly be using a *depth-first search* (DFS) of a graph, which at each step chooses to explore an arc leaving the most recently visited vertex (that still has unexplored arcs leaving it). Instead, a *breadth-first search* (BFS), always chooses an arc leaving the least recently visited vertex. See Figure A.1 for an example. (We use this graph in Section 2.4; see Figure 2.9.)

Depth-first search

Sometimes we refer to the DFS numbering produced by a depth-first search as the preorder numbering, since it gives the same numbering as a preorder traversal of the corresponding DFS tree D. Numbering the vertices by the order they became inactive produces a postorder numbering (same as the postorder traversal of D).

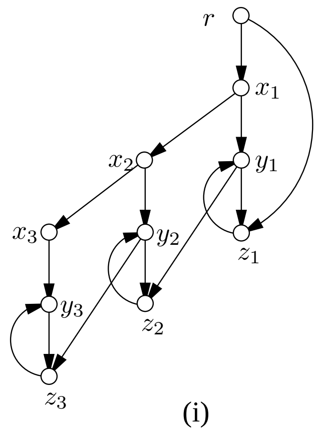
Consider any fixed DFS tree D of G. The next lemma describes a useful fact about DFS that we will use often.

Lemma A.1 [Tar72] *Let D be a DFS tree of $G = (V, A, r)$. If v and w are vertices in V such that $v \leq w$ (w.r.t. the DFS numbering), then any path from v to w must contain a common ancestor of v and w in D.*

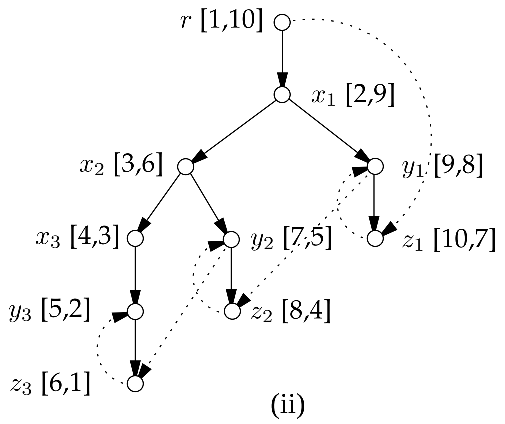
When it is clear from the context, we drop the D subscript from the notation $v \xrightarrow{*}_D w$, $v \xrightarrow{+}_D w$, $v \rightarrow_D w$ and $p_D(v)$. Let v and w be any vertices such that $v < w$. If $(v, w) \in A$ then by Lemma A.1, $v \xrightarrow{*} w$. Hence, if $v = p(w)$, (v, w) is a *parent arc* (or *tree arc*), otherwise it is a *forward arc*. If $(w, v) \in A$ and $v \xrightarrow{+} w$ then (w, v) is a *back arc* (or *cycle arc*). If $(w, v) \in A$ and v and w are unrelated then we call (w, v) a *cross arc*.

A.2 Models of computation

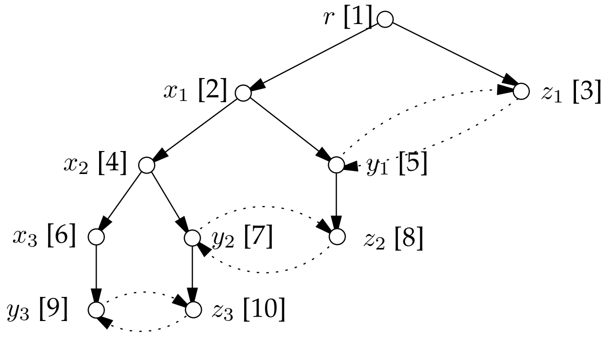
We will consider two standard models of computation that are used for the complexity analysis of algorithms; the *random-access machine* [AHD74] and the *pointer machine* [Tar83].



(i)



(ii)



(iii)

Figure A.1: Graph search. (i) Input graph. (ii) Depth-first search. The numbers inside each bracket correspond to the preorder and postorder numbers of a vertex. Non-tree arcs are dotted. (iii) Breadth-first search. The numbers correspond to the BFS numbering.

A.2.1 Random-access machine

A random-access machine (RAM) consists of a finite program, a finite collection of registers and a memory. The memory consists of an array of cells, each having a unique integer address. Both a memory cell and a register can store a single integer or real number of size bounded by the *word length* w, which is a parameter of the model. A typical assumption is that w is proportional to the logarithm of the size of the input. A RAM can perform in a single step one of the following tasks: execute a single arithmetic or logical operation using the contents of its registers, load the contents of a single memory cell into a register, or store the contents of a register to a memory cell. The load and store operations require that the address of the memory cell that will be accessed is stored in a register.

A.2.2 Pointer machine

A pointer machine differs from a RAM in the organization of the memory. The memory of a pointer machine consists of an extendable collection of nodes. Each node can store a fixed number of fields, and a field can store either a number or a pointer to a node. Creating a memory node takes one time step. The only way to access the contents of a node is by having in a register a pointer to that node, and accessing a node given that pointer also takes one time step.

A.2.3 Remarks

Clearly a pointer machine is less powerful than a RAM. One usual way in which algorithms exploit the power of a RAM is by performing address arithmetic; since the address of a cell is just an integer this address can be computed using arithmetic operations, which enables the access to that cell. This type of access is necessary in order to implement data structure tools like hashing. A pointer machine lacks this capability; there, the only way to access a memory node is by following pointers that connect to it.

The RAM model is realistic enough (if we overlook the capability of manipulating arbitrary real numbers in constant time) in the sense that the set of operations that it allows can be implemented by standard programming languages such as C. Still, besides its theoretical interest, the study of pointer machine algorithms has also practical value since pointer machines are capable of simulating functional programming languages such as LISP [BAG92].

A.3 Input format

The algorithms we consider receive as their input a flowgraph G, and sometimes a tree T. We assume that G is represented in the *adjacency list* format; each vertex v of the graph is associated with a linked list that contains the successors of v in G and a linked list that contains the predecessors of v in G. Note that predecessors can be computed from successors in linear time, and vice versa. Hence it suffices to have either a successor-list or a predecessor-list representation of G. This description of G requires $O(m + n)$ space, where (throughout this dissertation) m is the number of arcs and n is the number of vertices of G. A tree T is represented similarly since it is a special kind of flowgraph.

References

[AB04] 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.

[ABB] S. Allesina, A. Bodini, and C. Bondavalli. Secondary extinctions in ecological networks: Bottlenecks unveiled. *Ecological Modeling*. In press.

[ABHS00] F. S. Annexstein, K. A. Berman, T. Hsu, and R. P. Swaminathan. A multi-tree routing scheme using acyclic orientations. *Theor. Comput. Sci.*, 240(2):487–494, 2000.

[AC72] F. E. Allen and J. Cocke. Graph theoretic constructs for program control flow analysis. Technical Report IBM Res. Rep. RC 3923, IBM T.J. Watson Research Center, 1972.

[AFPB01] 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.

[AHD74] A. V. Aho, J. E. Hopcroft, and J. D.Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Series in Computer Science and Information Processing, MA, 1974.

[AHLT99] S. Alstrup, D. Harel, P. W. Lauridsen, and M. Thorup. Dominators in linear time. *SIAM Journal on Computing*, 28(6):2117–32, 1999.

[AHU76] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. On finding lowest common ancestors in trees. *SIAM Journal on Computing*, 5(1):115–32, 1976.

[AL96] S. Alstrup and P. W. Lauridsen. A simple and optimal algorithm for finding immediate dominators in reducible graphs. Technical Report DIKU TOPPS D-260, Dept. of Computer Science, U. Copenhagen, 1996.

[ALT96] S. Alstrup, P. W. Lauridsen, and M. Thorup. Dominators in linear time. Technical Report DIKU TOPPS D-320, Dept. of Computer Science, U. Copenhagen, 1996.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.

[AU77] A. V. Aho and J. D. Ullman. *Principles of Compilers Design*. Addison-Wesley, 1977.

[BAG92] A. M. Ben-Amram and Z. Galil. On pointers versus addresses. *Journal of the ACM*, 39(3):617–648, 1992.

[BCD$^+$02] M. A. Bender, R. Cole, E. D. Demaine, M. Farach-Colton, and J. Zito. Two simplified algorithms for maintaining order in a list. *Lecture Notes in Computer Science*, 2461:152–164, January 2002.

[BFC00] M. A. Bender and M. Farach-Colton. The LCA problem revisited. In *Proc. 4th Latin American Symp. on Theoretical Informatics*, volume 1776 of *Lecture Notes in Computer Science*, pages 88–94. Springer-Verlag, 2000.

[BGK$^+$] A. L. Buchsbaum, L. Georgiadis, H. Kaplan, A. Rogers, R. E. Tarjan, and J. R. Westbrook. Linear-time pointer-machine algorithms for least common ancestors, MST verification, and dominators. In preparation.

[BKRW98a] A. L. Buchsbaum, H. Kaplan, A. Rogers, and J. R. Westbrook. Linear-time pointer-machine algorithms for least common ancestors, MST verification, and dominators. In *Proc. 30th ACM Symp. on Theory of Computing*, pages 279–88, 1998.

[BKRW98b] A. L. Buchsbaum, H. Kaplan, A. Rogers, and J. R. Westbrook. A new, simpler linear-time dominators algorithm. *ACM Transactions on Programming Languages and Systems*, 20(6):1265–96, 1998. Corrigendum appeared in 27(3):383-7, 2005.

[CAD] CAD Benchmarking Lab. ISCAS'89 benchmark information. `http://www.cbl.ncsu.edu/www/CBL_Docs/iscas89.html`.

[CFR+91] 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.

[CH05] R. Cole and R. Hariharan. Dynamic LCA queries on trees. *SIAM Journal on Computing*, 34(4):894–923, 2005.

[CHK] K. D. Cooper, T. J. Harvey, and K. Kennedy. A simple, fast dominance algorithm. Available online at `http://www.cs.rice.edu/∼keith/EMBED/dom.pdf`.

[CLR91] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Electrical Engineering and Computer Science Series. MIT Press, Cambridge, MA, 1991.

[CLY] S. Curran, O. Lee, and X. Yu. Finding four independent trees. Submitted paper. Preprint available at `http://www.ic.unicamp.br/ lee/4tree.pdf`.

[CM88] J. Cheriyan and S. N. Maheshwari. Finding nonseparating induced cycles and independent spanning trees in 3-connected graphs. *Journal of Algorithms*, 9:507–537, 1988.

[CR92] J. Cheriyan and J. H. Reif. Directed s-t numberings, rubber bands, and testing digraph k-vertex connectivity. In *Proc. 3rd ACM-SIAM Symp. on Discrete Algorithms*, pages 335–344, 1992.

[Die00] R. Diestel. *Graph Theory*. Springer-Verlag, New York, second edition, 2000.

[DRT92] B. Dixon, M. Rauch, and R. E. Tarjan. Verification and sensitivity analysis of minimum spanning trees in linear time. *SIAM Journal on Computing*, 21(6):1184–92, 1992.

[DS87] 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.

[Edm70] J. Edmonds. Submodular functions, matroids, and certain polyhedra. *Combinatorial Structures and their Applications*, pages 69–81, 1970.

[Edm72] J. Edmonds. Edge-disjoint branchings. *Combinatorial Algorithms*, pages 91–96, 1972.

[ET76] S. Even and R. E. Tarjan. Computing an st-numbering. *Theoretical Computer Science*, 2(3):339–344, 1976.

[Fis72] M. J. Fischer. Efficiency of equivalence algorithms. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 153–168. Plenum Press, New York, 1972.

[FOW87] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9:319–349, July 1987.

[FW94] M. L. Fredman and D. E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *Journal of Computer and System Sciences*, 48:533–51, 1994.

[Gab90] H. N. Gabow. Data structures for weighted matching and nearest common ancestors with linking. In *Proc. 1st ACM-SIAM Symp. on Discrete Algorithms*, pages 434–43, 1990.

[Gab00] H. N. Gabow. Path-based depth-first search for strong and biconnected components. *Information Processing Letters*, 74:107–114, 2000.

[GBT84] H. N. Gabow, J. L. Bentley, and R. E. Tarjan. Scaling and related techniques for geometry problems. In *Proc. 16th ACM Symp. on Theory of Computing*, pages 135–43, 1984.

[GI91] Z. Galil and G. F. Italiano. Data structures and algorithms for disjoint set union problems. *ACM Computing Surveys*, 23(3):319–44, 1991.

[Gol01a] A. V. Goldberg. Shortest path algorithms: Engineering aspects. In *Proceedings of the International Symposium on Algorithms and Computation (ISAAC '01)*, volume 2223 of *Lecture Notes in Computer Science*, pages 502–513. Springer-Verlag, 2001.

[Gol01b] A. V. Goldberg. A simple shortest path algorithm with linear average time. In *Proceedings of the 9th European Symposium on Algorithms (ESA '01)*, volume 2161 of *Lecture Notes in Computer Science*, pages 230–241. Springer-Verlag, 2001.

[GPF04] A. Gal, C. W. Probst, and M. Franz. Complexity-based denial-of-service attacks on mobile code systems. Technical Report 04-09, School of Information and Computer Science, University of California, Irvine, 2004.

[GPF05] A. Gal, C. W. Probst, and M. Franz. Average case vs. worst case: Margins of safety in system design. In *Proceedings of the New Security Paradigms Workshop*, 2005. To appear.

[GT85] H. N. Gabow and R. E. Tarjan. A linear-time algorithm for a special case of disjoint set union. *Journal of Computer and System Sciences*, 30(2):209–21, 1985.

[GT04] L. Georgiadis and R. E. Tarjan. Finding dominators revisited. In *Proc. 15th ACM-SIAM Symp. on Discrete Algorithms*, pages 862–871, 2004.

[GT05] L. Georgiadis and R. E. Tarjan. Dominator tree verification and vertex-disjoint paths. In *Proc. 16th ACM-SIAM Symp. on Discrete Algorithms*, pages 433–442, 2005.

[GTW] L. Georgiadis, R. E. Tarjan, and R. F. Werneck. Finding dominators in practice. *Journal of Graph Algorithms and Applications (JGAA)*. Invited paper accepted for publication.

[GWT$^+$04] L. Georgiadis, R. F. Werneck, R. E. Tarjan, S. Triantafyllis, and D. I. August. Finding dominators in practice. In *12th Annual European Symposium on Algorithms*, pages 677–688, 2004.

[Har85] D. Harel. A linear algorithm for finding dominators in flow graphs and related problems. In *Proc. 17th ACM Symp. on Theory of Computing*, pages 185–194, 1985.

[HT84] D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal on Computing*, 13(2):338–55, 1984.

[HU74] M. S. Hecht and J. D. Ullman. Characterizations of reducible flow graphs. *Journal of the ACM*, 21(3):367–375, 1974.

[Huc95] A. Huck. Disproof of a conjecture about independent branchings in k-connected directed graphs. *Journal of Graph Theory*, 20(2):235–239, 1995.

[Huc99a] A. Huck. Independent branchings in acyclic digraphs. *Discrete Math*, 199:245–249, 1999.

[Huc99b] A. Huck. Independent trees and branchings in planar multigraphs. *Graphs and Combinatorics*, 15:211–220, 1999.

[HY97] G. Holloway and C. Young. The flow analysis and transformation libraries of Machine SUIF. In *Proceedings of the 2nd SUIF Compiler Workshop*, 1997.

[IMP] The IMPACT compiler. `http://www.crhc.uiuc.edu/IMPACT`.

[IR84] A. Itai and M. Rodeh. Three tree-paths. In *Proc. 25th IEEE Symp. on Foundations of Computer Science*, pages 137–147, 1984.

[IR88] A. Itai and M. Rodeh. The multi-tree approach to reliability in distributed networks. *Information and Computation*, 79(1):43–59, 1988.

[IZ89] A. Itai and A. Zehavi. Three tree-paths. *Journal of Graph Theory*, 13:175–188, 1989.

[Kin97] V. King. A simpler minimum spanning tree verification algorithm. *Algorithmica*, 18:263–70, 1997.

[KKT95] D. R. Karger, P. N. Klein, and R. E. Tarjan. A randomized linear-time algorithm to find minimum spanning trees. *Journal of the ACM*, 42(2):321–28, 1995.

[Knu71] D. E. Knuth. An empirical study of fortran programs. *Software Practice and Experience*, 1:105–133, 1971.

[KU76] J. B. Kam and J. D. Ullman. Global data flow analysis and iterative algorithms. *Journal of the ACM*, 23:158–171, 1976.

[LEC67] A. Lempel, S. Even, and I. Cederbaum. An algorithm for planarity testing of graphs. In *Proceddings International Symposium on Theory of Graphs*, pages 215–232. Gordon and Breach, 1967.

[LT79] 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.

[MS94] B. M. E. Moret and H. D. Shapiro. An empirical assessment of algorithms for constructing a minimum spanning tree. *DIMACS Monographs in Discrete Mathematics and Theoretical Computer Science*, 15:99–117, 1994.

[Muc97] S. S. Muchnick. *Advanced Compiler Design and Implementation*, chapter 14. Morgan-Kaufmann Publishers, San Francisco, CA, 1997.

[Ple91] J. Plehn. *Über die Existenz und das Finden von Subgraphen*. PhD thesis, University of Bonn, Germany, May 1991.

[PM72] P. W. Purdom, Jr. and E. F. Moore. Algorithm 430: Immediate predominators in a directed graph. *Communications of the ACM*, 15(8):777–778, 1972.

[Pou96] H. La Poutré. Lower bounds for the union-find and the split-find problem on pointer machines. *Journal of Computer and System Sciences*, 52:87–99, 1996.

[PW02] M. Poggi de Aragão and Renato F. Werneck. On the implementation of MST-based heuristics for the Steiner problem in graphs. In D. M. Mount and C. Stein, editors, *Proceedings of the Fourth International Workshop on Algorithm Engineering and Experiments (ALENEX'02)*, volume 2409 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag, 2002.

[RHS95] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 49–61, June 1995.

[RR94] G. Ramalingam and T. Reps. An incremental algorithm for maintaining the dominator tree of a reducible flowgraph. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 287–296, 1994.

[SB92] P. H. Sweany and S. J. Beaty. Dominator-path scheduling: A global scheduling method. In *Proceedings of the 25th International Symposium on Microarchitecture*, pages 260–263, 1992.

[Sha80] M. Sharir. Structural analysis: A new approach to flow analysis in optimizing compilers. *Computer Languages*, 5(3):141–153, 1980.

[SP81] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S.S. Muchnick and N.D. Jones, editors, *Program flow analysis: theory and applications*, pages 189–233. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[SPE] The Standard Performance Evaluation Corp. `http://www.spec.org/`.

[ST83] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26:362–391, 1983.

[SV88] B. Schieber and U. Vishkin. On finding lowest common ancestors: Simplification and parallelization. *SIAM Journal on Computing*, 17(6):1253–62, 1988.

[Tar72] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–59, 1972.

[Tar73] R. E. Tarjan. Testing flow graph reducibility. In *Proceedings of the Fifth Annual ACM Symposium on Theory of Computing*, pages 96–107, 1973.

[Tar74] R. E. Tarjan. Finding dominators in directed graphs. *SIAM Journal on Computing*, 3(1):62–89, 1974.

[Tar75] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, 1975.

[Tar76] R. E. Tarjan. Edge-disjoint spanning trees and depth-first search. *Acta Informatica*, 6(2):171–85, 1976.

[Tar79a] R. E. Tarjan. Applications of path compression on balanced trees. *Journal of the ACM*, 26(4):690–715, 1979.

[Tar79b] R. E. Tarjan. A class of algorithms which require nonlinear time to maintain disjoint sets. *Journal of Computer and System Sciences*, 18(2):110–27, 1979.

[Tar81] R. E. Tarjan. Fast algorithms for solving path problems. *Journal of the ACM*, 28(3):594–614, 1981.

[Tar83] R. E. Tarjan. *Data structures and network algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1983.

[TvL84] R. E. Tarjan and J. van Leeuwen. Worst-case analysis of set union algorithms. *Journal of the ACM*, 31(2):245–81, 1984.

[Vui80] J. Vuillemin. A unifying look at data structures. *Communications of the ACM*, 23(4):229–39, 1980.

[Whi87] R. W. Whitty. Vertex-disjoint paths and edge-disjoint branchings in directed graphs. *Journal of Graph Theory*, 11:349–358, 1987.

[Wir04] A. Wirth. Manuscript, 2004.