Dominators in Linear Time*

Stephen Alstrup[†] Dov Harel[‡] Peter W. Lauridsen[†]   Mikkel Thorup[†]

Abstract

A linear time algorithm is presented for finding dominators in control flow graphs.

Key words. Control flow analysis, dominators, algorithms.

AMS subject classifications. 68Q25, 68N20.

1   Introduction

Finding the dominator tree for a control flow graph is one of the most fundamental problems in the area of global flow analysis and program optimization [2, 3, 4, 5, 10, 15]. The problem was first raised in 1969 by Lowry and Medlock [15], where an $O(n^4)$ algorithm for the problem was proposed (as usual, n is the number of nodes and m the number of edges in a graph). The result has been improved several times (see e.g. [1, 2, 17, 20]), and in 1979 an $O(m\alpha(m,n))$ algorithm was found by Lengauer and Tarjan [14]. Finally, at STOC'85, Dov Harel [11] announced a linear time algorithm. Based on Harel's result, linear time algorithms have been found for many other problems (see e.g. [4, 5, 10]). Harel's description was, however, incomplete. In this paper, we give a complete description of a different and simpler linear time dominator algorithm.

The paper is divided as follows. In section 2 the main definitions are given. In section 3 we outline the Lengauer-Tarjan algorithm and in section 4 we give a linear time dominator algorithm. Finally an appendix is included, in which we briefly discuss dominators in the simpler case of reducible control flow graphs. Furthermore the appendix contains implementation details of the algorithm in section 4.

2   Definitions

A *control flow graph* is a directed graph $G = (V, E)$, with $|V| = n$ and $|E| = m$, in which $s \in V$ is a start node, from which all nodes in V are reachable through the edges in E (see e.g. [2]). If $(v, w) \in E$ we say that node v is a *predecessor* of node w and w is a *successor* of v. Node v *dominates* w if and only if all paths from s to w pass through v. Hence, if both u and v dominates w, one of u and v dominates the other. Thus the dominance relation is the reflexive and transitive closure of a unique tree T, rooted in s. The tree T is called the *dominator tree* of G. If v is the parent of w in the dominator tree, then v *immediately dominates* w, denoted as $idom(w) = v$.

3   Lengauer and Tarjan's algorithm

In this section we outline Lengauer and Tarjan's algorithm, since the idea behind our algorithm is to optimize subroutines used in this algorithm.

Lengauer and Tarjan's algorithm [14] runs in $O(m\alpha(m,n))$ time. Initially a Depth First Search (DFS) [19] is performed in the graph resulting in a DFS-tree $\mathcal{T}$, in which the nodes are assigned a DFS-number. In this paper we will not distinguish between a node and its DFS-number. The nodes are thus ordered such that $v < w$ if the DFS-number of v is smaller than the DFS-number of w.

The main idea of the Lengauer-Tarjan algorithm is first to compute the so called *semidominators*, $sdom(v)$, for each node $v \in V \setminus \{s\}$, as an intermediate step for finding dominators. The semidominator of a node v is an ancestor of v defined as

$$sdom(v) = min\{u | \text{a path } u, w_1, \ldots, w_k, v \text{ exists, where } w_i > v \text{ for all } i = 1, \ldots, k\}.$$

The semidominators are found by traversing the tree $\mathcal{T}$ in decreasing DFS-number order while maintaining a dynamic forest, $\mathcal{F}$, which is a subgraph of the DFS-tree $\mathcal{T}$. The following operations should be supported on $\mathcal{F}$:

- LINK(v, w): Adds the edge $(v, w) \in \mathcal{T}$ to $\mathcal{F}$. The nodes v and w are root nodes of trees in $\mathcal{F}$.

- EVAL(v): Finds the minimum *key* value of nodes on the path from v to the root of the tree in $\mathcal{F}$, to which v belongs[1].

- UPDATE(v, k): Sets $key(v)$ to be k, where the node v must be a singleton tree.

We will now give a more detailed description of the Lengauer-Tarjan algorithm. The forest $\mathcal{F}$ initially contains all nodes as singleton trees and the computation of semidominators is done as follows:

- Initially we set $key(v) = v$ for all nodes $v \in V$.

- The nodes are then visited in decreasing DFS-number order, i.e. v is visited before w if and only if $v > w$. When visiting a node v, we call UPDATE(v, k), where $k = min\{\text{EVAL}(w) | (w, v) \in E\}$.

- After visiting v, a call LINK(v, w) is made for all children w of v.

After running this algorithm we have $key(v) = sdom(v)$. The correctness of the algorithm (i.e. that when a node is updated, it is with the correct $sdom$-value), follows from the following theorem given by Lengauer and Tarjan [14, Theorem 4]:

Theorem 1 *For any node $v \neq s$, $sdom(v) = min(S_1 \bigcup S_2)$ where $S_1 = \{w | (w, v) \in E \land w < v\}$ and $S_2 = \{sdom(u) | u > v \land (w, v) \in E \land u \text{ is an ancestor of } w\}$.* $\square$

To see the connection between Theorem 1 and the algorithm above consider the visit of node v in the algorithm. Since EVAL$(w) = w$ for $w < v$, $S_1 = \{\text{EVAL}(w) | (w, v) \in E \land w < v\}$. To see that $S_2 = \{\text{EVAL}(w) | (w, v) \in E \land w > v\}$, note that if $u > v$, $(w, v) \in E$ and u is an ancestor of w in $\mathcal{T}$ then u and w have already been visited. Thus u is an ancestor of w in a tree in $\mathcal{F}$, so EVAL(w) includes $sdom(u)$.

Tarjan and Lengauer show that having found the semidominators, the immediate dominators can be found within the same complexity.

The EVAL-LINK operations in the algorithm are performed using a slightly modified version of Tarjan's UNION-FIND algorithm for disjoint sets [21]. Since n LINK and m EVAL operations are performed the complexity is $O(m\alpha(m, n))$. Thus a linear time algorithm can be obtained if the EVAL and LINK operations can be performed in $O(n + m)$ time.

4   A linear time algorithm

In this section we present a linear time dominator algorithm. The overall idea is to convert the on-line EVAL-LINK algorithm to an off-line algorithm by exploiting the fact that the tree resulting from LINK operations is known in advance. The inspiration for this stems from the linear UNION-FIND algorithm for disjoint sets by Gabow and Tarjan [9]. In the Gabow-Tarjan algorithm, the tree, T, resulting from all UNION operations is known in advance. More specifically this means that a UNION(v, w) operations is only permitted if the edge (v, w) is in T. The FIND queries are then defined as usual, whereas UNION(v, w) is

_____

[1] In [14] EVAL operations only include the root of the tree in case the root is the only node in the tree. We have given the definition above to avoid confusion, as it is this definition which will be used in our algorithm. The Lengauer-Tarjan algorithm presented here is therefore a slight modification of the original algorithm. More specifically the modification consists of performing the LINK(v, w) operation when v is visited instead of when w is visited.

defined as the union of the sets to which v and w belongs. The linear time is achieved by tabulating the behavior of UNION-FIND within small "microtrees" of size $O(\log n)$.

The original approach of Harel was to convert this linear UNION-FIND algorithm into an EVAL-LINK algorithm [11]. Roughly speaking, the basic idea was to define a new parameter of nodes, referred to as pseudo-dominator, which satisfy the following two conditions: (a) pseudo-dominators can be propagated in linear time, and (b) using pseudo-dominators we can compute semidominators in linear time. This approach had a couple of drawbacks, further elaborated upon in appendix B. Here we do not involve pseudo-dominators, but calculate the semidominators directly. We use, not only that we know the resulting tree-structure, but also that we know that the LINK operations come in reverse DFS-order. Instead of converting the linear UNION-FIND algorithm, we end up using it as a black box. Moreover, the information needed for tabulating EVAL is found using Fredman and Willard's Q-heaps [7], which were not available at the time of [11]. Finally, our choice of microtrees leads to simpler calculations.

To be more specific, we will construct an algorithm which performs the n LINK and UPDATE operations interspersed with m EVAL operations in $O(n + m)$ time. As an intermediate step we will first present a simple algorithm with complexity $O(n \log n + m)$ and then extend it to handle a special kind of update. Next we present a faster algorithm for the case, in which the DFS-tree $\mathcal{T}$ is a path. The combination of these algorithms gives a fast algorithm for trees with few leaves. We then limit the number of leaves in $\mathcal{T}$ by removing small subtrees. Finally we apply the algorithm recursively to the small subtrees in order to get subtrees small enough for tabulation.

4.1 An $O(n \log n + m)$ algorithm

We consider a forest, $\mathcal{F}$, of trees. Recall that to each node a *key* is associated, which initially contains the DFS-number of the node. Let T_v denote the tree in $\mathcal{F}$, to which v belongs. We will use the term *selfcontained* for nodes, for which $EVAL(v) = key(v)$. Hence a node v is selfcontained if all ancestors of v in T_v have key values $\geq key(v)$. Note that the definition implies that all root nodes in $\mathcal{F}$ are selfcontained. A node v stops being selfcontained when T_v is linked to a root node u, for which $key(u) < key(v)$.

Lemma 1 *Let $nsa(v)$ denote the nearest selfcontained ancestor of v.*

(a) *For any node $v \in V$, we have $EVAL(v) = key(nsa(v))$.*

(b) *For any node pair $u, v \in V$, if $nsa(v) = u$ at some point in the Lengauer-Tarjan algorithm, then $nsa(v) = nsa(u)$ in the remainder of the algorithm.*

Proof.

(a) By the definition of selfcontained nodes $key(nsa(v))$ is the least *key* value of nodes on the path from $nsa(v)$ to the root of T_v. By the same definition, if nodes with *key* values $< key(nsa(v))$ were on the path from v to $nsa(v)$ in T_v, the node with least depth among these nodes would be selfcontained.

(b) By definition $nsa(u)$ is the first selfcontained node on the path from u to the root of T_u. The fact that $nsa(v) = u$ implies that $T_v = T_u$ and that all nodes on the path from v to u in T_v have *key* values $> key(u)$. By the definition of UPDATE none of these nodes will change *key* values again. The node $nsa(v)$ will therefore always be the first selfcontained node on the path from u to the root of T_u. □

By the second part of lemma 1 we can represent the *nsa*-relation efficiently by using disjoint sets. Let each selfcontained node, u, be the *canonical element* of the set $\{v | nsa(v) = u\}$. By the first part of lemma 1 an $EVAL(v)$ operation is then reduced to finding the canonical element of the set to which v belongs, hence $EVAL(v) = key(SetFind(v))$.

When a $LINK(u, v)$ operation is performed, the node v will no longer be the root of T_v. Therefore a set of nodes in T_v may stop being selfcontained. Let A be this set of nodes. A node, $w \in A$, is the canonical element of a set containing nodes, whose EVAL values change from $key(w)$ to $key(u)$ by lemma 1. We can thus maintain the structure by unifying the sets associated with nodes in A with the set associated with u.

To find the set A, a heap, supporting HeapFindMax, HeapExtractMax and HeapUnion (e.g. [6, 22]), is associated with each root of a tree in $\mathcal{F}$. Each heap contains the selfcontained nodes in the tree (see figure 1).
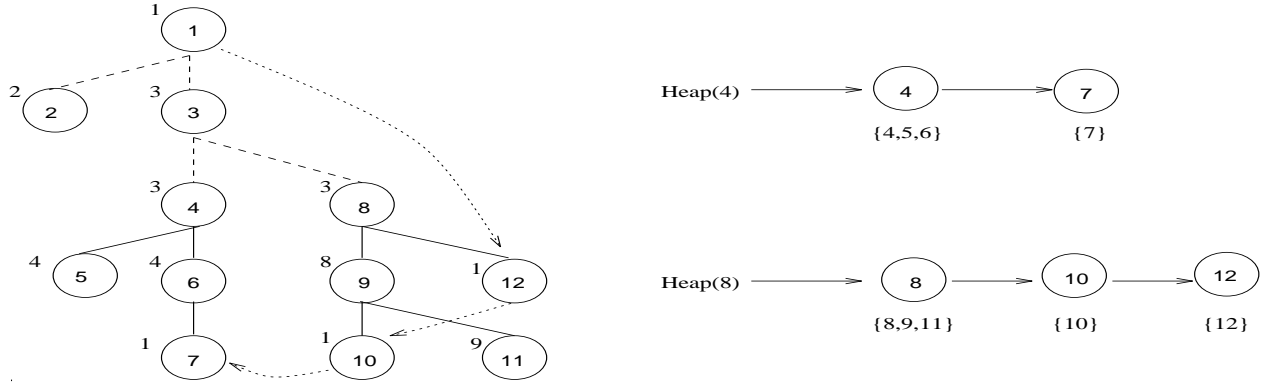
3

Figure 1: To the left a sample DFS-tree in which the nodes are labeled by their DFS-numbers is given. The full lines indicate the part of the tree, which has been linked, hence the node to be processed is node "3". The dotted arrows are graph edges. The numbers beside the nodes are their *key* values. To the right the two non-trivial heaps containing selfcontained nodes are illustrated as lists. Below the selfcontained nodes the sets associated with them are listed.

The set A can then be found by repeatedly extracting the maximum element from the heap associated with v until the maximum element of this heap is $\leq key(u)$.

The algorithm LINK(u, v) is thus

- **While not** $Empty(Heap(v))$ **and** $key(FindMax(Heap(v))) > key(u)$ **do**
-    $w := ExtractMax(Heap(v))$;
-     $SetUnion(u, w)$; /* The canonical element of the resulting set is u */
- **od**;
- $Heap(u) := HeapUnion(Heap(u), Heap(v))$;

Lemma 2 *The algorithm presented performs the n LINK and UPDATE operations interspersed with m EVAL operations in $O(m + n \log n)$ time.*

Proof. At most $O(n)$ *HeapExtract, HeapFindMax* and *HeapUnion* operations are performed. Each of these operations can be done in $O(\log n)$ time using an ordinary heap (e.g. [6, 22]). Since the tree structure is known in advance, the set operations can be computed in linear time using the result from [9][2]. It will however suffice to use a simple disjoint set algorithm which rearranges the smallest of the two sets.□

4.2   Decreasing roots

In section 4.4 we will need the ability to decrease the *key* value of a node, while it is the root of a tree. We will therefore extend the algorithm from the previous section to handle the $DecreaseRoot(v, k)$ operation, which sets $key(v) = k$, where v is the root of T_v. The $DecreaseRoot(v, k)$ operation should be done in constant time.

Assume that a $DecreaseRoot(v, k)$ operation has been performed. In analogy with the LINK operation from the previous section, this may imply that some selfcontained nodes in T_v are no longer selfcontained. We should therefore remove such nodes from the heap and unify the sets associated with them, with the set associated with v, as was done in the LINK operation. However, in the algorithm from the previous section the root node v is the maximum element in the heap associated with it. In order to remove nodes from the heap we would therefore first have to remove v, which would require $O(\log n)$ time. We should note that since the heap returns maximum values the usual *decreasekey* operation for heaps cannot be used. We can

---

[2] If this result is used, the $SetUnion$ operation should be changed according to the description given earlier in this section. More specificly the call would be $SetUnion(parent(w), w)$ and the canonical element of the resulting set would be the canonical element of the set $parent(w)$ belongs to.

4

however take advantage of the fact that the root node will always be the maximum element in the heap it belongs to. It is therefore not necessary to explicitly insert the root into the heap before it is linked to its parent. The $DecreaseRoot(v, k)$ operation is performed as follows.

- **While not** $Empty(Heap(v))$ **and** $key(FindMax(Heap(v))) > k$ **do**
- $\quad w := ExtractMax(Heap(v));$
- $\quad SetUnion(v, w);$
- **od**;
- $key(v) := k;$

Lemma 3 *We can perform d DecreaseRoot and n LINK and UPDATE operations interspersed with m EVAL operations in $O(n \log n + m + d)$ time.*

Proof. We change the algorithm from the previous section by postponing the insertion of a root node, r, into $Heap(r)$, until r is linked to its parent. This has no effect on the complexity of EVAL and UPDATE operations stated in lemma 2. Since each node may be deleted from a heap at most once, the total number of $ExtractMax$ and $SetUnion$ operations invoked by LINK and $DecreaseRoot$ is still $O(n)$. The cost of these operations can therefore be charged to the LINK operations. Since the remaining operations invoked by $DecreaseRoot$ are done in constant time each, the additional complexity of the d $DecreaseRoot$ operations is $O(d)$.□

4.3 A linear time algorithm for paths

We consider the situation in which the tree $\mathcal{T}$ is a path. Recall that in the algorithms from the previous two subsections we needed a heap to order selfcontained nodes. The property which distinguishes paths from trees in this context is that this ordering is induced by the path. More specificly, any pair u, v of selfcontained nodes on the part of the path, which has been linked, are ordered such that $key(v) \geq key(u)$ if and only if $depth(v) \leq depth(u)$. To perform LINK operations on a path we can therefore use the algorithm from the previous section, where the heap is replaced by a stack. The algorithm for the operation LINK(u, v) on a path is thus.

- **While not** $StackEmpty$ **and** $key(StackTop) > key(u)$ **do**
- $\quad w := StackPop;$
- $\quad SetUnion(u, w);$/* The canonical element of the resulting set is u */
- **od**;
- $StackPush(u);$

An EVAL operation on a path is performed in analogy with the previous section, hence EVAL$(v) = key(SetFind(v))$.

Lemma 4 *If the tree $\mathcal{T}$ is a path, we can perform the n LINK and UPDATE operations and the m EVAL operations in $O(n + m)$ time.*

Proof. The stack operations are done in linear time since each node will only be on the stack once. By using the result from [9] the set operations are performed in amortized constant time. We should note that the result from [9] is more general than necessary and that it is possible to construct a simpler linear time algorithm for set operations on paths.□

4.4 A faster algorithm for trees with few leaves

We can take advantage of the linear time algorithm from the previous section by using it on the paths in $\mathcal{T}$. More specificly let R be the tree obtained by substituting each path in $\mathcal{T}$, which consist of (at least two) nodes with at most one child, by an artificial node. We will refer to such paths as *I-paths* and the artificial nodes as *I-nodes*. The correspondence between R and the forest $\mathcal{F}$ is the following:

- When the node with largest depth on an I-path is linked to its child, c, in $\mathcal{F}$, the *I-node* is linked to c in R.

5

- When the node with least depth on an I-path is linked to its parent, p, in $\mathcal{F}$, the *I-node* is linked to p in R.

We will use the result from section 4.2 for nodes in R and the result from the previous section for nodes on I-paths. The above correspondence means that EVAL queries on nodes in R correspond to EVAL queries in $\mathcal{F}$ if, for any I-path P, $key(\textit{I-node}(P))$ is the least key value on the part of P which has been linked. In other words we use $\textit{I-node}(P)$ to represent the minimum selfcontained node on P in R. During the processing of an I-path P the key value of $\textit{I-node}(P)$ should thus be properly updated. This is done by invoking a $DecreaseRoot(\textit{I-node}(P), k)$ operation each time a new minimum key value k is found on P.

The EVAL queries on nodes on an I-path, P, will be correct, as long as the node with least depth on P has not yet been linked to its parent. We can therefore construct an interface between R and I-paths as follows. We associate a pointer, *I-root*, with each node on an I-path. The pointer is initially set to be NULL and when the node with least depth on an I-path is linked to its parent p, we set $\textit{I-root}(v) = p$, for all nodes v belonging to the I-path. The algorithm for EVAL(v) is thus (we use subscripts to distinguish between the structures EVAL operations are performed in):

- **if** v belongs to an I-path P **then**
-    **if** $\textit{I-root}(v) =$NULL **then return** EVAL$_P(v)$
-    **else return** $min\{$EVAL$_P(v),$EVAL$_R(\textit{I-root}(v))\}$
- **else return** EVAL$_R(v)$;

Lemma 5 *Let l denote the number of leaves in $\mathcal{T}$. We can perform m EVAL and n LINK and UPDATE operations in $O(l \log l + m + n)$ time.*

Proof. The I-paths are processed in linear time by lemma 4. Since the I-paths have been contracted the tree R contains $O(l)$ nodes. Thus by lemma 3, R can be processed in time $O(l \log l + m + n + d)$, where d is the number of $DecreaseRoot$ operations. The number of $DecreaseRoot$ operations is however bounded by the number of nodes on I-paths.$\square$

4.5 Reducing to small subtrees

From lemma 5 we have that the EVAL-LINK algorithm can be performed effectively on trees with few leaves. However the number of leaves is only bounded by the number of nodes. To reduce the number of leaves in $\mathcal{T}$, subtrees of size $\leq \log n$ can be removed. We will refer to such subtrees as *S-trees*. Assume that all S-trees have been removed from the tree $\mathcal{T}$. Then each leaf in the remaining tree must be a node in $\mathcal{T}$ with at least $\log n$ descendants. Thus the remaining tree has at most $n/\log n$ leaves. By lemma 5 we can therefore perform amortized constant time EVAL, LINK and UPDATE operations in the remaining tree.

We now show how to process the S-trees. Recall that the LINK operations are performed in decreasing DFS-number order. This implies that EVAL operations of nodes in S-trees induced by nodes outside, will only take place at a time when all links have been performed inside the structure. Furthermore the links inside S-trees are performed successively, hence each S-tree can be processed independently. Analogously with I-paths we can associate a pointer *S-root* with each node in each S-tree, which points to the parent of the root of the S-tree after the LINK between the root and its parent has been performed. Then an EVAL operation on a node v in an S-tree becomes:

$$\begin{array}{ll} \text{EVAL}_S(v), & \text{if } \textit{S-root}(v)=\text{NULL} \\ min\{\text{EVAL}_S(v),\text{EVAL}(\textit{S-root}(v))\}, & \text{otherwise} \end{array}$$

To perform EVAL, LINK and UPDATE operations inside an S-tree we could use lemma 2. Alternatively we could repeat the removal of subtrees on the S-trees because of the independent nature of S-trees. Let $T(m, n, a)$ denote the time it takes to support the the m EVAL and n LINK and UPDATE operations in the Lengauer-Tarjan algorithm within subtrees of $\mathcal{T}$ each of size $\leq a$. For example, the construction of lemma 2, gives $T(m, n, a) = O(m + n \log a)$.

Lemma 6 $T(m, n, a) = O(m + n) + T(m, n, \log a)$.

Proof. Choose the S-trees to be of size at most $\log n$. Then, in the upper tree, we have at most $n/\log n$ leaves, so, by lemma 5, the cost of the LINKs and UPDATEs there is $O(n)$. EVAL queries to the upper tree have a constant cost by lemma 2. An EVAL query to an S-tree may propagate to the root via the S-root-pointer, but retains constant time complexity.□

Since $T(m, n, 1) = O(m + n)$, repeating the above recurrence $\log^* n$ times, we immediately get

$$T(m, n, n) = O((m + n)\log^* n)$$

However, in this paper, we only need to repeat it twice, giving

Corrolary 1 $T(m, n, n) = O(m + n) + T(m, n, \log\log n)$.□

In the next subsection, we will show that $T(m, n, \log\log n) = O(m+n)$, implying a linear time algorithm for finding dominators.
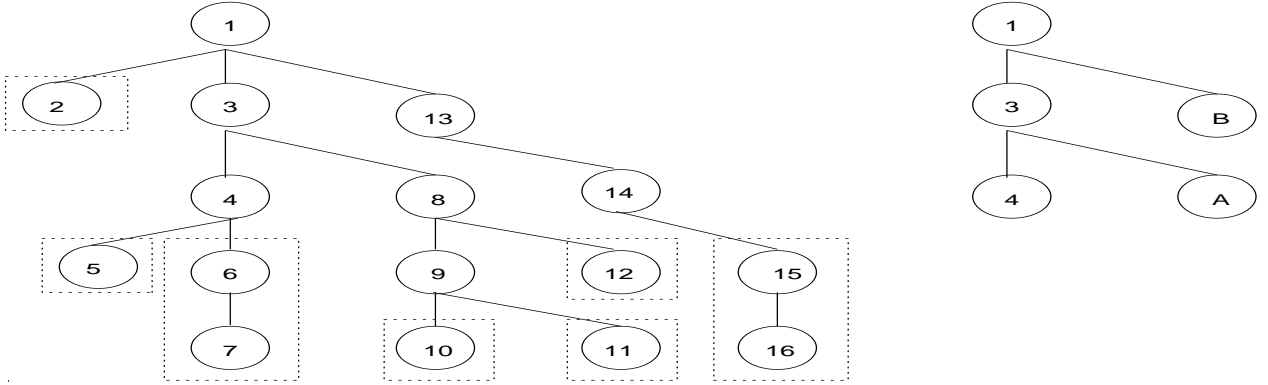

Figure 2: To the left a sample DFS-tree is given. The boxes indicate S-trees of size ≤ 2. To the right the reduced tree is given. The nodes "A" and "B" are replacing I-paths. Note that if S-trees of size $\leq \log_2 16 = 4$ were removed, the tree would be reduced to a single node representing the I-path $1, 3, 8$.

In figure 2 the division of a tree into I-paths and S-trees in one level is illustrated.

4.6 Tabulation of small trees

In this section we show how to perform constant time EVAL, UPDATE and LINK operations on trees of size $\leq \log\log n$, henceforth denoted as *microtrees*. We will do this by constructing a table containing EVAL values for all possible forest permutations. We first show how to compute such a table assuming that a superset of *sdom* values is known for each microtree. Following that we show how to choose this superset. Combining these results we show that the microtrees can be processed in linear time. Finally we give the theorem, which completes the dominator algorithm. We start out by giving a lemma by Fredman and Willard [7].

Lemma 7 *The Q-heap performs insertion, deletion, and search operations in constant time and accommodates as many as* $(\log n)^{1/4}$ *items given the availability of* $O(n)$ *time and space for preprocessing and word size* $\geq \log n$. □

For a set M' of different values we define the rank of a value $x \in M'$ as the number of values $< x$ in M'.

Lemma 8 *If the rank of sdom-values for all nodes in a tree of size k are known in advance, we can preprocess the tree in $O(k)$ time, such that all EVAL operations can be done in constant time.*

Proof. Let r denote the root of the tree. We traverse the tree top-down and set EVAL(r)=r and for each node $v \neq r$ set EVAL(v)=min($key(v)$,EVAL($parent(v)$)).□

Theorem 2 *Assume that to each microtree M we are given a set of values M', where $|M'| = O(|M|)$ and that for all $UPDATE(v, k)$ operations, $v \in M \Rightarrow k \in M'$. Assume also that the order in which LINK operations occur is known. It is then possible to perform constant time EVAL, LINK and UPDATE operations, given the availability of $O(n)$ time and space for preprocessing and word size $\geq \log n$.*

Proof. In order to perform constant time EVAL queries we tabulate all possible forest configurations as follows:

We construct each possible tree of size $\leq \log \log n$. Since in general there are at most $O(2^k)$ trees of size k (all trees of size k can be uniquely represented by a bitmap of size $2k$), there are at most $\log n$ such trees. For each of these trees we construct the $\log \log n$ possible ways the nodes in the tree can be partially linked. Finally for each of these forests we construct copies holding all possible permutations of ranks to nodes. In each of these forests we compute the EVAL-value for each node. We then construct a table which outputs the computed EVAL-values. By lemma 8 this computation can be done in a time proportional to the number of nodes in the trees. The number of nodes is the product of the number of trees ($\log n$), the number of LINK's ($\log \log n$), the number of rank permutations ($(C_1 * \log \log n)^{\log \log n}$) and the number of nodes in each tree ($\log \log n$), thus the number of nodes is (C_i are constants):

$\log n * \log \log n * (C_1 * \log \log n)^{\log \log n} * \log \log n =$
$\log n^{C_2} * (\log \log n)^2 * \log \log n^{\log \log n} \leq$
$\log n^{C_3} * \log \log n^{\log \log n} =$
$\log n^{C_3} * \log n^{\log \log \log n} =$
$\log n^{C_3 + \log \log \log n} = O(n)$.

To store each forest, the forest table from [9], which require $\log \log n$ space, can be used. The rank of each node require $\log \log \log n$ space. If we attach a new number to each node inside the forest we can identify each node using $\log \log \log n$ space. Hence each entry to the table requires $\log \log n + \log \log \log n * \log \log n + \log \log \log n$ space, which will fit into a computer word of size $\geq \log n$. The size of the table is thus $O(n)$ (for details see appendix A.2).

Given this table each microtree can be processed as follows: For each microtree we sort the sets M' of size $O(\log \log n)$ in linear time using lemma 7. The *key* value of each node is replaced by their rank in M', which simply is an index into the sorted set. To carry out the operations given a microtree, we first compute the table entry for the tree without any links. The EVAL operations are done by looking up the table and the LINK and UPDATE operations are done by updating the entry (again we refer to appendix A.2 for details). Finally in order to perform UPDATE and EVAL operations we need a table which maps *key* values to ranks and vice versa. Since all *key* values are $< n$, this table only requires $O(n)$ space.□

Theorem 2 requires a superset M' of *sdom* values for nodes in a microtree M. The next lemma shows how M' can be chosen.

Lemma 9 *Let $M' = M \bigcup \{min\{(EVAL(w))| \; (w, v) \in E \wedge w \notin M\}| \; v \in M\}$. For all $v \in M$ we have that $sdom(v) \in M'$.*

Proof. The lemma is obviously true in case $sdom(v) \in M$. Assume therefore that $u = sdom(v) \notin M$ and that $u \notin M'$. By the definition of semidominators a path $u = w_0, w_1, \ldots, w_{k-1}, w_k = v$ exists where $w_i > v$ for $i = 1, \ldots, k-1$. Let w_j be the last node on the path not in M. Since $u \notin M'$ the node w_{j+1} must have a predecessor x for which $\text{EVAL}(x) < u$. This means that a path exists from a node u', with $u' < u$, to w_{j+1} on which all nodes except u' are $> v$. This path can be concatenated with the path $w_{j+1}, \ldots, w_k, v$, contradicting that $sdom(v) = u$.□

We now complete the microtree algorithm by showing how to compute the sets M' of lemma 9.

Theorem 3 *Let M be a microtree of size $\leq \log \log n$. Each EVAL, UPDATE and LINK operation inside M in the Lengauer-Tarjan algorithm can be performed in constant time, given the availability of $O(n)$ time and space for preprocessing and word size $\geq \log n$.*

Proof. By theorem 2 and lemma 9 we only need to show how to compute the sets M' defined in lemma 9 in $O(|M'|)$ time. We will show this by induction on the visits of microtrees. Recall that the Lengauer-Tarjan

algorithm visits nodes in decreasing DFS-number order. When the first microtree is reached all nodes with larger DFS-numbers have thus been processed. By corollary 1 EVAL queries on processed nodes outside microtrees can be done in constant time. Furthermore all nodes with smaller DFS-numbers will at this stage be singleton trees. The EVAL queries required in lemma 9 can thus be performed in constant time for the first microtree. Given an arbitrary microtree M we can therefore assume that constant time EVAL queries can be performed in microtrees containing nodes with larger DFS-numbers than the nodes in M. For nodes not in microtrees, we can compute the EVAL values needed in lemma 9 in constant time By the same arguments as above. By induction this is also the case for nodes in previously visited microtrees.

Finally we should note that in the proof of theorem 2, $O(n)$ space was used for the table, which maps *key* values to ranks for a microtree. Since the microtrees are computed independently, this space can be re-used, so that the overall space requirement is $O(n)$.$\Box$

We can now combine the results of this section in the following theorem.

Theorem 4 *The EVAL, LINK and UPDATE operations in the Lengauer-Tarjan algorithm can be performed in linear time.*

Proof. Follows directly from corollary 1 and theorem 3.$\Box$

5 Concluding remarks

A linear time algorithm has been presented for finding dominators. The result, as presented, is purely theoretical, in the sense that Fredman and Willard's Q-heaps require that $n \geq 2^{12^{2^0}}$ [7]. Some of our ideas may still be of practical relevance. If, for example, we take corollary 1, giving a rather simple linear time reduction to subtrees of size at most $\log \log n$, and then use lemma 2 within each of these, we get a simple $O(m + n \log \log \log n)$ algorithm, which in practice may be competitive with the one of Lengauer and Tarjan [14].

In the following appendices we present the pseudo-code of our algorithm, some details of the tabulation, and a very simple linear time dominator algorithm for the common special case of reducible control-flow graphs.

References

[1] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. On finding lowest common ancestors in trees. In *Annual ACM Symposium on the theory of computing (STOC)*, volume 5, pages 115–132, 1973.

[2] A.V. Aho and J.D. Ullman. *The Theory of Parsing, Translation and Compiling*, volume II. Prentice-Hall, Englewood Cliffs, N.J., 1972.

[3] A.V. Aho and J.D. Ullman. *Principles of compiler design*. Addison-Wesley, Reading, MA, 1979.

[4] G. Bilardi and K. Pingali. A framwork for generalized control dependence. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 291–300, 1996.

[5] R. Cytron, J. Ferrante, B.K. Rosen, M.N. Wegman, and F.K. Zadek. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Programming Language Systems (TOPLAS)*, 13(4):451–490, 1991.

[6] J.R. Driscoll, H.N. Gabow, R. Shrairman, and R.E. Tarjan. Relaxed heaps: An alternative to fibonacci heaps with application to parallel computation. *Communcation of the ACM (C.ACM)*, 31(11):1343–1354, 1988.

[7] M.L. Fredman and D.E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *Journal of Computer and System Sciences*, 48(3):533–551, 1994.

[8] H.N. Gabow. Data structure for weighted matching and nearest common ancestors with linking. In *Annual ACM-SIAM Symposium on discrete algorithms (SODA)*, volume 1, pages 434–443, 1990.

[9] 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*, 30:209–221, 1985.

[10] G.R. Gao and V.C. Sreedhar. A linear time algorithm for placing ϕ-nodes. In *ACM SIGPLAN-SIGACT Symposium on the Principles of Programming Languages (POPL)*, pages 62–73, 1995.

[11] D. Harel. A linear time algorithm for finding dominators in flow graphs and related problems. In *Annual ACM Symposium on theory of computing (STOC)*, volume 17, pages 185–194, 1985.

[12] D.E. Knuth. *The art of programming*, volume 1. Addison-Wesley, 1968.

[13] P.W. Lauridsen. Dominators. Master's thesis, Department of Computer Science, University of Copenhagen, 1996.

[14] T. Lengauer and R.E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Programming Languages Systems (TOPLAS)*, 1:121–141, 1979.

[15] E.S. Lowry and C.W. Medlock. Object code optimization. *Communication of the ACM (C.ACM)*, 12(1):13–22, 1969.

[16] R. Ochranova. Finding dominators. *Fundamentals (or Foundations) of Computation Theory*, 4:328–334, 1983.

[17] P.W. Purdom and E.F. Moore. Immediate predominators in a directed graph. *Communication of the ACM (C.ACM)*, 15(8):777–778, 1972.

[18] G. Ramalingam and T. Reps. An incremental algorithm for maintaining the dominator tree of a reducible flowgraph. In *Annual ACM Symposium on Principles of Programming Languages (POPL)*, volume 21, pages 287–298, 1994.

[19] R.E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

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

[21] R.E. Tarjan. A class of algorithms which require nonlinear time to maintain disjoint sets. *Journal of computer and system sciences*, 18(2):110–127, 1979.

[22] J. Vuillemin. A data structure for manipulating priority queues. *Communcation of the ACM (C.ACM)*, 21(4):309–315, 1978.

[23] N. Wirth. *Programming in modula-2(3rd corr.ed)*. Springer–verlag, Berlin, New York, 1985.

A   Implementation details

This section contains details about the algorithm presented in the paper. The main algorithm is described in section A.1 and details about the construction and use of microtables are described in section A.2.

A.1   The main algorithm

We assume that a DFS-search has been performed in the graph. The I-paths are removed from the tree in the following way:
The child pointer of the parent to the first node and the parent pointer of the child of the last node are removed. Instead an *I-node* is inserted (see figure 3). The *I-node* is numbered by a unique number larger than n. Furthermore the I-paths are numbered by a number > 0.

The algorithm uses the following arrays in which the DFS-number of nodes are used as indices (the arrays marked * are also used in the Lengauer-Tarjan algorithm):
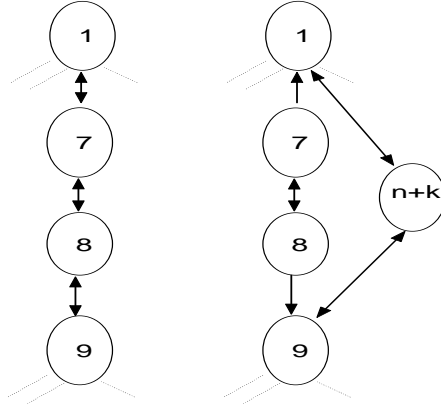
Figure 3: An I-path and the representation of the I-path in the tree. Both child and parent pointers are illustrated

- $pred(v)^*$: The set of nodes w such that $(w, v) \in E$.
- $parent(v)^*$: The parent of v in the DFS-tree. To simplify the EVAL operation we set $parent(v) = 0$ if $v = 0$.
- $child(v), sibling(v)$: Pointer to the first child and first sibling of v respectively in the DFS-tree.
- $I\text{-}path(v)$: If v does not belong to an I-path, $I\text{-}path(v) = 0$. Otherwise $I\text{-}path(v)$ contains the number of the I-path to which v belongs.
- $S\text{-}tree(v)$: True if v belongs to an S-tree.
- $microtree(v)$: True if v belongs to a microtree.
- $root(v)$: This field is defined for nodes in S-trees, microtrees and I-paths. Before the root of the structure has been linked to its parent, p, $root(v) = 0$. Afterwards $root(v)$ contains the number of p.
- $first(v)$: If v belongs to an I-path, this field contains the number of the first node on the I-path.
- $stack(v)$: If v is the first node on an I-path, this field contains the stack used for the I-path.
- $microroot(v)$: If v belongs to a microtree then $microroot(v)$ is the number of the root of the microtree.

- $key(v)^{*3}$: After the semidominator of v has been computed $key(v)$ is the number of the semidominator of v. Initially $key(v) = v$.
- $bucket(v)^*$: The set of nodes whose semidominator is v.
- $dom(v)^*$: A number which will eventually be the number of the immediate dominator of v.

The main algorithm is a slight modification of the Lengauer-Tarjan algorithm:

Begin
 $construct\,microtable$; /* This procedure computes the microtable */
 for $v := 1$ **to** n **do** $bucket(v) := \emptyset$;
 $v := n$;
 While $v > 1$ **do begin**
  if $microtree(v)$ **then begin**
   $microdominator(v, microroot(v))$; /* see below */
   $v := microroot(v) - 1$;
  end else begin
   For each $w \in pred(v)$ **do begin**
    $k :=$**EVAL**(w);
    if $k < key(v)$ **then UPDATE**(v, k);
   end;
   /* The remainder of the algorithm computes dominators */
   /* from semidominators and is analogous to [14] */

---

[3] In the Lengauer-Tarjan algorithm this array is called $semi$.

```
        For each child w of v do LINK(v, w);
        bucket(key(v)) := bucket(key(v)) ⋃ {v};
        While bucket(parent(v)) ≠ ∅ do begin
          bucket(parent(v)) := bucket(parent(v))\{w};
          k := EVAL(w)
          if k < key(w) then dom(w) := k
          else dom(w) := parent(v);
        end;
        v := v − 1;
      end; /* While /*
    end; /* While /*
    for v := 2 to n do
      if dom(v) ≠ key(v) then dom(v) := dom(dom(v));
      else dom(v) := key(v);
End;
```

Procedure *microdominator*(*v, root* : *integer*);

The *microdominator* procedure is analogous to the main algorithm. The only real difference is that EVAL, LINK and UPDATE operations are replaced by microEVAL, microLINK and microUPDATE operations. Furthermore there are no I-paths in a microtree.

For the EVAL and LINK operations we need the following additional fields:

- *heap*(*v*): A heap associated with *v*.
- *I-node*(*v*): If *v* is the root of an I-path then *I-node*(*v*) is the number of the node which represents the I-path.

Function EVAL(*v*: integer):integer;
```
 begin
   if v = 0 then EVAL:=∞
   else if microtree(v) then EVAL:=min(EVAL(root(v),microEVAL(v)))
   else if I-path(v) or S-tree(v) then EVAL:=min(EVAL(root(v)),key(SetFind(v)))
   else EVAL:=key(SetFind(v));
 end;
```

Procedure LINK(*v, w*: integer);
```
 begin
   if microtree(w) then /* w is the root of a microtree */
     For each u in the microtree to which w belongs do root(u) := v
   else if S-tree(w) and not S-tree(v) then /* w is the root of an S-tree */
     For each u in the S-tree to which w belongs do root(u) := v
   else if I-path(v) > 0 then begin
     if first(v) = v then Init-I-path(v, w) /*see below */
     else if I-path(w) = I-path(v) then begin
       S := stack(first(v));
       While not StackEmpty(S) and key(StackTop(S)) > key(v) do begin
         u := StackPop(S);
         SetUnion(v, u);
       end;
       if key(I-node(v)) > key(v) then DecreaseRoot(I-node(v), key(v));
       StackPush(v, S);
     end;
   end else if I-path(w) > 0 then begin /* the path is fully linked */
     For each u on the I-path do root(u) := v;
     /* Add I-node(w) to heap(I-node(w)) */⁴
     LINK(v, I-node(w));
   end else begin /* Neither v or w is on an I-path */
     While not Empty(heap(w)) and key(HeapFindMax(heap(w))) > key(v) do begin
       u := HeapExtractMax(heap(w));
```

---

⁴The *I-node* has not been a member of the heap while the I-path has been processed. The pseudo code for this operation is omitted to improve program clarity, as it involves creating a dummy heap and performing a *HeapUnion* operation on the dummy heap and *heap*(*I-node*(*w*)).

```
        SetUnion(v, u);
      end;
      HeapUnion(heap(v), heap(w));
    end;
  end;


Procedure Init-I-path(v, w: integer);
/* v is the first node on an I-path and should be linked to its child w */
begin
  CreateStack(S);
  stack(v) := S;
  StackPush(v, S);
  While not Empty(heap(w)) and key(HeapFindMax(heap(w))) > key(v) do begin
    w := HeapExtractMax(heap(w));
    SetUnion(I-node(v), w);
  end;
  heap(I-node(v)) := heap(w);
  key(I-node(v)) := key(v);
end;


Procedure DecreaseRoot(v, k: integer);
begin
  While not Empty(heap(v)) and key(HeapFindMax(heap(v))) > k do begin
    w := HeapExtractMax(heap(v));
    SetUnion(v, w);
  end;
  key(v) := k;
end;


Procedure UPDATE(v, k: integer);
begin
  key(v) := k;
end;
```

A.2   The microalgorithm

In the proof of theorem 3 the forest table from [9] was suggested to store the forests. The forest from [9] support any ordering of the LINK operations, whereas in the dominator algorithm the links are performed in decreasing DFS-number order. We can therefore simplify the representation by using the DFS-traversal to represent each tree. More specifically we start at the root and use a bitmap in which '1' means that an edge is followed down in the tree and a '0' means that we move to the parent of the current node. The tree traversal is finished when a '0' is encountered while the root is the current node. As a special case this means that a single node tree is represented by the bitmap "0". The mapping is illustrated in figure 4. Instead of representing the LINK's explicitly we can save the number of nodes in the tree, which at some point in time has been processed by the algorithm. Since the size of the forests can differ we also need to save the size of each tree. Finally the *key* and EVAL values of the nodes can be saved in order of the DFS-traversal. The bitmap of an entry can thus have the following configuration [SIZE‖TREE‖KEYS‖EVAL‖LINK], where SIZE and and LINK are blocks of $\log \log \log n$ bits, EVAL and KEYS uses SIZE bits and TREE uses (2*SIZE-1) bits.

To construct the entry of a microtree we traverse it in DFS-order and set the bits of TREE and SIZE accordingly. The KEYS are initialized to the rank of the DFS-numbers and LINK is initalized to 0. A microLINK operation is performed by incrementing the LINK value and the microUPDATE(v, k) operation is done by replacing the value of v in the entry with k.

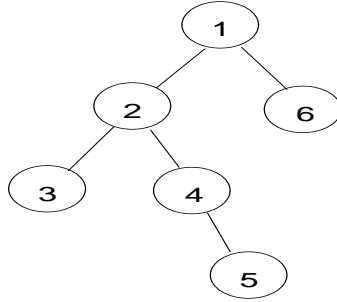The pseudo code of the microalgorithm is rather tedious and therefore omitted.

Figure 4: A sample tree labeled by DFS-numbers. The bitmap of the tree is '11011000100'.

B Relation to Harel's algorithm

The proof of Theorem 1 of [11] which is omitted, employs a linear time table construction using a variant of dynamic programming. The details of this construction are beyond the scope of this paper.

Harel's original value propagation required the construction of supersets of the sets of sdom values for all microtrees in a separate phase, in order to presort the values. The main drawback of the technique is that it leads to a rather complicated case analysis, and checking correctness is pretty tedious. In fact the original value propagation algorithm in [11] contains an error (more precisely Theorem 3b in [11] is false as stated and a concrete counterexample is given in [13, Section 4, p. 12]).

The algorithm described in this paper avoids the above problems by replacing the tables required to prove Theorem 1 of [11] by the use of Fredman and Willard's priority queues. This technique is more general and allows us to propagate semi-dominator (sdom) values on a per microset basis, just prior to computing the exact semi-dominators values for all members of a microset.

C Algorithms for reducible graphs

The problem of finding dominators in reducible graphs has been investigated in several papers (e.g. [1, 16, 18]). The reason why reducible graphs are considered is that the control flow graphs of certain programming languages (e.g. Modula-2 [23]) are reducible. A graph is reducible if the edges can be partitioned into two disjoint sets E' and E'' so that

- The graph induced by the edges in E' is acyclic.

- For all edges $(v, w) \in E''$, w dominates v.

Since the edges E'' have no influence on the dominance relation the problem of finding dominators in reducible graphs is analogous to finding dominators in acyclic graphs. In this section we therefore assume that graphs are acyclic.

C.1 The former algorithm is not linear

In 1983 Ochranova [16] gave an algorithm which is claimed to have complexity $O(m)$[5]. Unfortunately the paper does not contain a complexity analysis. In order to disprove the complexity of the algorithm it is therefore necessary to outline the behavior of the algorithm. For an acyclic graph we have the following facts:

---

[5] Citation: "At least no counterexample was found."

(a) If a node, x, has a single predecessor, y, then $idom(x) = y$.

(b) If each of the successors of a node x has more than one predecessor then no node is dominated by x.

Since at least one successor of the start node s will satisfy the condition in (a) the dominators can be found by starting at s and using the two facts interchangeably as follows:

1. If (a) is true for a successor, v, of the current node, w, then set $idom(v) = w$ and the current node to v.

2. If (b) is true for all successors of the current node w then merge w and $idom(w)$ (by unifying their successor and predecessor sets respectively). Set the current node to be the merged node.

In order for the algorithm to be linear the detection of whether (a) is true in 1 should have constant time complexity. Furthermore the merge of two nodes in 2, which involves union of two sets which are not disjoint, should also have constant time complexity. The authors are not aware of a general algorithm with the above properties.

C.2 A linear time algorithm

In this section we give a simple linear time algorithm for finding dominators in reducible graphs. The algorithm is constructed by combining new techniques [8] with previously presented ideas (see e.g. [1, 18]). In other words the algorithm is a compilation.

The computation is divided into two main steps as follows.

1. The graph $G = (V, E')$ is acyclic and can therefore be topologically sorted [12] ensuring that if $(v, w) \in E'$ then v has a lower topological number than w.

2. Now the dominator tree T can be constructed dynamically. Set s to be the root of the dominator tree T and process the nodes from $V \backslash \{s\}$ in increasing topological order as follows. (Notice that the part of T, built so far, is used for determining $idom$ for the rest of the nodes.)

   - Let $W = \{v | (v, w) \in E'\}$ be the set of predecessors of w in G and let A be the set of ancestors in T to all nodes in W. The node $idom(w)$ is then the node in A with the largest depth in T. Hence $idom(w)$ can be computed by repeatedly deleting two arbitrary nodes from W and inserting the nearest common ancestor (nca) of these nodes into the set W until the set contains only one node.
   - After computing $idom(w)$ the edge $(w, idom(w))$ is added to T.

The only unspecified part of the algorithm is the computation of nca in a tree T which grows under the addition of leaves. In [8] an algorithm is given which processes nca and addition of leaves in constant time per operation.

Theorem 5 *The algorithm above computes the dominator tree for a reducible control flow graph with n nodes and m edges in $O(n + m)$ time.*

Proof. Step 1 in the algorithm has complexity $O(n + m)$. In step 2 each node is visited and each edge can result in a query about nca in T, so at most m nca-queries are performed, which establishes the complexity. $\square$