Finding Dominators in Practice*

Loukas Georgiadis[1], Renato F. Werneck[1], Robert E. Tarjan[1,2], Spyridon Triantafyllis[1], and David I. August[1]

[1] Dept. of Computer Science, Princeton University, Princeton NJ, 08544, USA
[2] Hewlett-Packard, Palo Alto, CA

Abstract. The computation of dominators in a flowgraph has applications in program optimization, circuit testing, and other areas. Lengauer and Tarjan [17] proposed two versions of a fast algorithm for finding dominators and compared them experimentally with an iterative bit vector algorithm. They concluded that both versions of their algorithm were much faster than the bit-vector algorithm even on graphs of moderate size. Recently Cooper et al. [9] have proposed a new, simple, tree-based iterative algorithm. Their experiments suggested that it was faster than the simple version of the Lengauer-Tarjan algorithm on graphs representing computer program control flow. Motivated by the work of Cooper et al., we present an experimental study comparing their algorithm (and some variants) with careful implementations of both versions of the Lengauer-Tarjan algorithm and with a new hybrid algorithm. Our results suggest that, although the performance of all the algorithms is similar, the most consistently fast are the simple Lengauer-Tarjan algorithm and the hybrid algorithm, and their advantage increases as the graph gets bigger or more complicated.

1 Introduction

A flowgraph $G = (V, A, r)$ is a directed graph with $|V| = n$ vertices and $|A| = m$ arcs such that every vertex 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. Our goal is to find for each vertex v in V the set $Dom(v)$ of all vertices that dominate v. Certain applications require computing the *postdominators* of G, defined as the dominators in the graph obtained from G by reversing all arc orientations.

Compilers use dominance information extensively during program analysis and optimization, for such diverse goals as natural loop detection (which enables a host of optimizations), structural analysis [20], scheduling [22], and the computation of dependence graphs and static single-assignment forms [10]. Dominators are also used to identify pairs of equivalent line faults in VLSI circuits [7].

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 [5]. A direct implementation of this solution

* L. Georgiadis, R. F. Werneck and R. E. Tarjan partially supported by the Aladdin project, NSF Grant No. CCR-9626862.

has $O(mn^2)$ worst-case time. Purdom and Moore [18] gave a straightforward algorithm with complexity $O(mn)$. It consists of performing a search in $G - v$ for all $v \in V$ (v obviously dominates all the vertices that become unreachable). Improving on previous work by Tarjan [23], Lengauer and Tarjan [17] proposed an $O(m\alpha(m,n))$-time algorithm, where $\alpha(m,n)$ is an extremely slow-growing functional inverse of the Ackermann function. Alstrup et al. [6] gave a linear-time solution for the random-access model; a simpler solution was given by Buchsbaum et al. [8]. Georgiadis and Tarjan [12] achieved the first linear-time algorithm for the pointer-machine model.

Experimental results for the dominators problem appear in [17, 8, 9]. In [17] 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 show that Purdom-Moore [18] is only competitive for graphs with fewer than 20 vertices, and that a bit-vector implementation of the iterative algorithm, by Aho and Ullman [4], is 2.5 times slower than LT for graphs with more than 100 vertices. Buchsbaum et al. [8] show that their claimed linear-time algorithm has low constants, being only about 10% to 20% slower than LT for graphs with more than 300 vertices. This algorithm was later shown to have the same time complexity as LT [12], and the corrected version is more complicated (see Corrigendum of [8]). Cooper et al. [9] present a space-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).

In this paper we cast the iterative algorithm into a more general framework and explore the effects of different initializations and processing orderings. 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 describe a new algorithm that combines LT with the iterative algorithm and is very fast in practice. Finally, we present a thorough experimental analysis of various algorithms using real as well as artificial data. We did not include linear-time algorithms in our study; they are significantly more complex and thus unlikely to be faster than LT in practice.

2 Algorithms

The *immediate dominator* of a vertex v, denoted by $idom(v)$, is the unique vertex $w \neq v$ that dominates v and is dominated by all vertices in $Dom(v) - v$. The (immediate) *dominator tree* is a directed tree I rooted at r and formed by the edges $\{(idom(v), v) \mid v \in V - r\}$. A vertex w dominates v if and only if w is a proper ancestor of v in I [3], so it suffices to compute the immediate dominators. Throughout this paper the notation "$v \xrightarrow{*}_F u$" means that v is an ancestor of u in the forest F and "$v \xrightarrow{+}_F u$" means that v is a proper ancestor of u in F. We omit the subscript when the context is clear. Also, we denote by $pred(v)$ the set of predecessors in G of vertex v. Finally, for any subset $U \subseteq V$ and a tree T, $\mathrm{NCA}(T, U)$ denotes the nearest common ancestor of $U \cap T$ in T.

2.1 The Iterative Algorithm

Clearly $Dom(r) = \{r\}$. For each of the remaining vertices, the set of dominators is the solution to the following data-flow equations:

$$Dom'(v) = \Big(\bigcap_{u \in pred(v)} Dom'(u) \Big) \cup \{v\}, \ \forall\ v \in V - r. \tag{1}$$

Allen and Cocke [5] showed that one can iteratively find the maximal fixed-point solution $Dom'(v) = Dom(v)$ for all v. Typically the algorithm either initializes $Dom'(v) \leftarrow V$ for all $v \neq r$, or excludes uninitialized $Dom'(u)$ sets from the intersection in (1). Cooper et al. [9] observe that the iterative algorithm does not need to keep each Dom' set explicitly; it suffices to maintain the transitive reduction of the dominance relation, which is a tree T. The intersection of $Dom'(u)$ and $Dom'(w)$ is the path from $\mathrm{NCA}(T, \{u, w\})$ to r. Any spanning (sub)tree S of G rooted at r is a valid initialization for T, since for any $v \in S$ only vertices in $r \stackrel{*}{\rightarrow}_S v$ can dominate v.

It is known [19] that the dominator tree I is such that if w is a vertex in $V - r$ then $idom(w) = \mathrm{NCA}(I, pred(w))$. Thus the iterative algorithm can be interpreted as a process that modifies a tree T successively until this property holds. The number of iterations depends on the order in which the vertices (or edges) are processed. Kam and Ullman [15] show that certain dataflow equations, including (1), are solved in up to $d(G, D) + 3$ iterations when the vertices are processed in reverse postorder with respect to a DFS tree D. Here $d(G, D)$ is the *loop connectedness of G with respect to D*, the largest number of back edges found in any cycle-free path of G. When G is reducible [13] the dominator tree is built in one iteration, because v dominates u whenever (u, v) is a back edge.

The running time per iteration is dominated by the time spent on NCA calculations. If they are performed naïvely (ascending the tree paths until they meet), then a single iteration costs $O(mn)$ 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 $d(G, D) \leq 3$ [16], and it is reasonable to expect that few NCA calculations will require $O(n)$ time. If T is represented as a dynamic tree [21], the worst-case bound per iteration is reduced to $O(m \log n)$, but the implementation becomes much more complicated.

Initializations and vertex orderings. Our base implementation of the iterative algorithm (IDFS) starts with $T \leftarrow \{r\}$ and processes the vertices in reverse postorder with respect to a DFS tree, as done in [9]. This requires a preprocessing phase that executes a DFS on the graph and assigns a postorder number to each vertex. Initializing T as a DFS tree is bad both in theory and in practice because it causes the back edges to be processed, 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 4 shows, this method is often (but not always) faster than IDFS.

Finally, we note that there is an ordering σ of the edges which is optimal with respect to the number of iterations that are needed for convergence. If we initialize $T = \{r\}$ and process the edges according to σ, then after one iteration we will have constructed the dominator tree. We are currently investigating if such an ordering can be found efficiently.

2.2 The Lengauer-Tarjan Algorithm

The Lengauer-Tarjan algorithm starts with a depth-first search on G from r and assigns preorder numbers to the vertices. The resulting DFS tree D is represented by an array *parent*. For simplicity, we refer to the vertices of G by their preorder number, so $v < u$ means that v has a lower preorder number than u. The algorithm is based on the concept of *semidominators*, which give an initial approximation to the immediate dominators. A path $P = (u = v_0, v_1, \ldots, v_{k-1}, v_k = v)$ in G is a *semidominator path* if $v_i > v$ for $1 \leq i \leq k-1$. The semidominator of v is defined as $sdom(v) = \min\{u \mid \text{there is a semidominator path from } u \text{ to } v\}$.

Semidominators and immediate dominators are computed by finding minimum *sdom* values on paths of D. Vertices are processed in reverse preorder, which ensures that all the necessary values are available when needed. The algorithm maintains a forest F such that when it needs the minimum $sdom(u)$ on a path $p = w \xrightarrow{+} u \xrightarrow{*} v$, w will be the root of a tree in F containing all vertices on p (in general, the root of the tree containing v is denoted by $root_F(v)$). Every vertex in V starts as a singleton in F. Two operations are defined on F: link(v) links the tree rooted at v to the tree rooted at $parent[v]$; eval(v) returns a vertex u of minimum *sdom* among those satisfying $root_F(v) \xrightarrow{+} u \xrightarrow{*} v$.

Every vertex w is processed three times. First, $sdom(w)$ is computed and w is inserted into a bucket associated with vertex $sdom(w)$. The algorithm processes w again after $sdom(v)$ has been computed, where v satisfies $parent[v] = sdom(w)$ and $v \xrightarrow{*} w$; then it finds either the immediate dominator or a *relative dominator* of w (an ancestor of w that has the same immediate dominator as w). Finally, immediate dominators are derived from relative dominators in a preorder pass.

With a simple implementation of link-eval (using path compression but not balancing), the LT algorithm runs in $O(m \log n)$ time [25]. With a more sophisticated linking strategy that ensures that F is balanced, LT runs in $O(m\alpha(m, n))$ time [24]. We refer to these two versions as SLT and LT, respectively.

Implementation issues. Buckets have very specific properties in the Lengauer-Tarjan algorithm: (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 be combined into a single array *bucket*.

In [17], Lengauer and Tarjan process $bucket[parent[w]]$ at the end of the iteration that deals with w. 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 $parent[w] = sdom(w)$ is not inserted into any bucket because we already know that $idom(w) = parent[w]$.

2.3 The SEMI-NCA algorithm

SEMI-NCA is a new hybrid algorithm for computing dominators that works in two phases: the first computes $sdom(v)$ for all $v \in V - r$ (as in LT), and the second builds I incrementally, using the fact that for any vertex $w \neq r$, $idom(w) = \text{NCA}(I, \{parent[w], sdom(w)\})$ [12]. In the second phase, for every w in preorder we ascend the I-tree path from $parent[w]$ to r until we meet the first vertex x such that $x \leq sdom(w)$. Then we have $x = idom(w)$. With this implementation, 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 $sdom(v)$ is usually a good approximation to $idom(v)$.

SEMI-NCA is simpler than LT in two 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. Second, buckets are no longer necessary because the vertices are processed in preorder in the second phase. With the simple implementation of link-eval (which is faster in practice), this method (SNCA) runs in $O(n^2 + m \log n)$ worst-case time. We note that Gabow [11] presents a rather complex procedure that computes NCAs in total linear time on a tree that grows by adding leaves. This implies that the second phase of SEMI-NCA can run in $O(n)$ time, but it is unlikely to be practical.

3 Worst-Case Behavior

This section briefly describes families of graphs that elicit the worst-case behavior of the algorithms we implemented. Figure 1 shows graph families that favor particular methods against the others. Family itworst(k) is a graph with $O(k)$ vertices and $O(k^2)$ edges for which IDFS and IBFS need to spend $O(k^4)$ time. Family idfsquad(k) has $O(k)$ vertices and $O(k)$ edges and can be processed in linear time by IBFS, but IDFS needs quadratic time; ibfsquad(k) achieves the reverse effect. Finally sncaworst(k) has $O(k)$ vertices and $O(k)$ edges and causes SNCA, IDFS and IBFS to run in quadratic time. Adding any (y_i, x_k) would make SNCA and IBFS run in $O(k)$ time, but IDFS would still need $O(k^2)$ time. We also define the family sltworst(k), which causes worst-case behavior for SLT [25]. Because it has $O(k)$ back edges, the iterative methods run in quadratic time.

4 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,

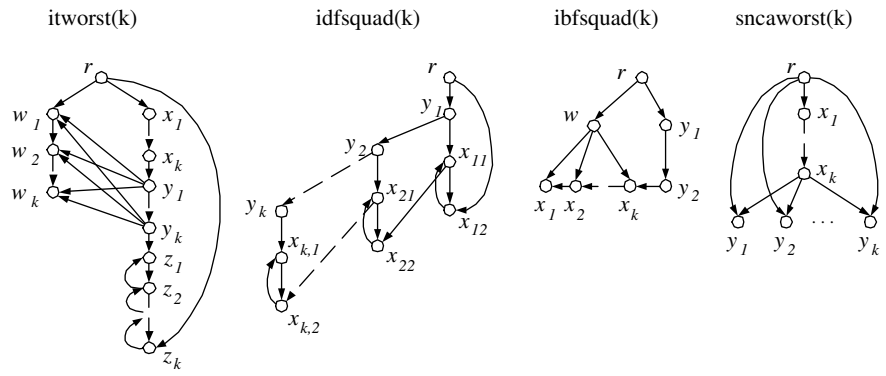

Fig. 1. Worst-case families.

however, "sophisiticated" algorithms tend to be harder to code and to have higher constants, so one might prefer other alternatives. 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 1 to n. Within reason, we made all implementations as efficient and uniform as we could. The source code is available from the authors upon request.

The code was compiled using `g++` v. 3.2.2 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 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. 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, but they do include allocating and deallocating the arrays used by each particular algorithm.

Instances. We used control-flow graphs produced by the SUIF compiler [14] from benchmarks in the SPEC'95 suite [2] and previously tested by Buchsbaum et al. [8] in the context of dominator analysis. We also used control-flow graphs created by the IMPACT compiler [1] 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 edges removed.[3] We also ran the algorithms on circuits from VLSI-testing applications [7] obtained from the ISCAS'89 suite [26] (all 50 graphs were considered a single class).

Finally, we tested eight instances that do not occur in any particular application related to dominators. Five are the worst-case instances described in Section 3, and the other three are large graphs representing speech recognition finite state automata (also used by Buchsbaum et al. [8]).

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. For each series, Table 1 shows the total number of graphs (g) and the average number of vertices and edges (n and m). As a reference, we report the average time (in microseconds) of a simple breadth-first search (BFS) on each graph. Times for computing dominators are given as multiples of BFS.

In absolute terms, all algorithms are reasonably fast: none was more than seven times slower than BFS. 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 faster than LT, but often slower than SLT. Both variants (IDFS and IBFS) usually have very similar behavior, although one method is occasionally much faster than the other (series 145.fppp and 256.bzip2 are good examples). Always within a factor of four of BFS, SNCA and SLT are the most consistently fast methods in the set.

By measuring the total 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. For each class, Table 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. Deviations, however, are higher. Together, these facts suggest that iterative methods are faster than other methods for small instances, but slower when size increases.

Figure 2 confirms this. Each point represents the mean relative running times for all graphs in the IMPACT class 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

[3] These edges appear in optimizing compilers due to superblock formation, and are produced much more often by IMPACT than by SUIF.

Table 1. Complete series: number of graphs (g), average number of vertices (n) and edges (m), and average time per graph (in microseconds for BFS, and relative to BFS for other methods). 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**
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. Times relative to BFS: geometric mean and geometric standard deviation. The best result in each row is marked in bold.

	IDFS		IBFS		LT		SLT		SNCA	
CLASS	MEAN	DEV	MEAN	DEV	MEAN	DEV	MEAN	DEV	MEAN	DEV
CIRCUITS	5.89	1.19	6.17	1.42	6.71	1.18	4.62	1.15	**4.40**	1.14
SUIF-FP	2.49	1.44	**2.34**	1.58	3.74	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**	1.77	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

IMPACTS and IMPACTP; for SUIF, which produces somewhat simpler graphs, iterative methods remained competitive even for larger sizes.

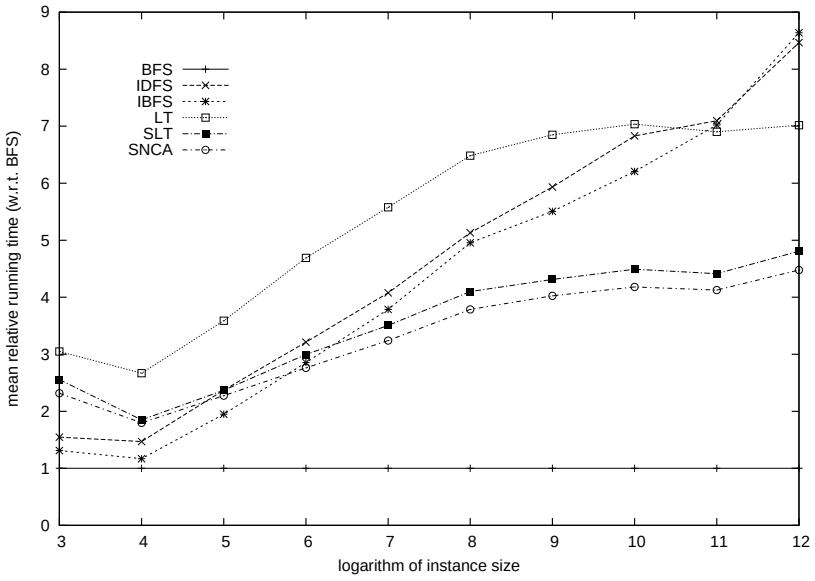

Fig. 2. Times for IMPACT instances within each size. Each point represents the mean relative running time (w.r.t. BFS) for all instances with the same value of $\lceil \log_2(n+m) \rceil$.

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

Table 3 helps explain the relative performance of the methods with three 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.

Table 3. Percentage of vertices that have their parents as semidominators (SDP%), average number of iterations and number of comparisons per vertex.

	SDP	ITERATIONS		COMPARISONS PER VERTEX				
CLASS	(%)	IDFS	IBFS	IDFS	IBFS	LT	SLT	SNCA
CIRCUITS	76.7	2.8000	3.2000	32.6	39.3	12.0	9.9	8.9
IMPACT	73.4	2.0686	1.4385	30.9	28.0	15.6	12.8	11.1
IMPACTP	88.6	2.0819	1.5376	30.2	32.2	15.5	12.3	10.9
IMPACTS	73.4	2.0686	1.4385	24.8	23.4	13.9	11.2	9.5
SUIF-FP	67.7	2.0000	1.6817	12.3	15.9	10.3	8.3	6.8
SUIF-INT	63.9	2.0009	1.6659	14.9	17.2	11.2	8.6	7.2

The next two columns show the average number of iterations performed by IDFS 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 last five columns show how many times on average a vertex is compared to other vertices (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, so their competitiveness results mainly from smaller constants. For example, 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 4 shows the number of vertices and edges, the time for BFS (in microseconds), and times for computing dominators (as multiples of BFS). The first five entries represent the worst-case familes described in Section 3. The last three graphs 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 now LT comes relatively close. Given enough vertices, the asymptotically better behavior of LT starts to show.

Table 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	EDGES	TIME	IDFS	IBFS	LT	SLT	SNCA
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**
itworst	401	10501	34	6410.5	6236.8	9.2	**4.7**	**4.7**
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**

5 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 have similar overall performance in practice. The tree-based iterative algorithms are the easiest to code and use less memory than the other methods, which makes them perform particularly well on small, simple graphs. Even on such instances, however, we did not observe the clear superiority of the original tree-based algorithm reported by Cooper et al. (our variants were not consistently better either). Both versions of LT and the hybrid algorithm 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 [17]. The simple version of LT and hybrid were the most consistently fast algorithms in practice; since the former is less sensitive to pathological instances, it should be the method of choice.

Acknowledgements. We thank Adam Buchsbaum for providing us the SUIF and speech recognition graphs and Matthew Bridges for his help with IMPACT. We also thank the anonymous referees for their helpful comments.

References

1. The IMPACT compiler. `http://www.crhc.uiuc.edu/IMPACT`.
2. The Standard Performance Evaluation Corp. `http://www.spec.org/`.
3. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.
4. A. V. Aho and J. D. Ullman. *Principles of Compilers Design*. Addison-Wesley, 1977.
5. 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.

6. S. Alstrup, D. Harel, P. W. Lauridsen, and M. Thorup. Dominators in linear time. *SIAM Journal on Computing*, 28(6):2117–32, 1999.
7. 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.
8. 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 to appear.
9. 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`.
10. 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.
11. H. N. Gabow. Data structures for weighted matching and nearest common ancestors with linking. In *Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*, pages 434–443, 1990.
12. L. Georgiadis and R. E. Tarjan. Finding dominators revisited. In *Proc. 15th ACM-SIAM Symp. on Discrete Algorithms*, pages 862–871, 2004.
13. M. S. Hecht and J. D. Ullman. Characterizations of reducible flow graphs. *Journal of the ACM*, 21(3):367–375, 1974.
14. G. Holloway and C. Young. The flow analysis and transformation libraries of Machine SUIF. In *Proceedings of the 2nd SUIF Compiler Workshop*, 1997.
15. J. B. Kam and J. D. Ullman. Global data flow analysis and iterative algorithms. *Journal of the ACM*, 23:158–171, 1976.
16. D. E. Knuth. An empirical study of FORTRAN programs. *Software Practice and Experience*, 1:105–133, 1971.
17. 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.
18. 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.
19. 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.
20. M. Sharir. Structural analysis: A new approach to flow analysis in optimizing compilers. volume 5, pages 141–153, 1980.
21. D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26:362–391, 1983.
22. 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.
23. R. E. Tarjan. Finding dominators in directed graphs. *SIAM Journal on Computing*, 3(1):62–89, 1974.
24. R. E. Tarjan. Applications of path compression on balanced trees. *Journal of the ACM*, 26(4):690–715, 1979.
25. R. E. Tarjan and J. van Leeuwen. Worst-case analysis of set union algorithms. *Journal of the ACM*, 31(2):245–81, 1984.
26. The CAD Benchmarking Lab, North Carolina State University. ISCAS'89 benchmark information. `http://www.cbl.ncsu.edu/www/CBL_Docs/iscas89.html`.