Dominators in Directed Graphs: A Survey of Recent Results, Applications, and Open Problems

Loukas Georgiadis Nikos Parotsidis
Department of Computer Science & Engineering, University of Ioannina, Greece
Email: {loukas,nparotsi}@cs.uoi.gr

***Abstract*—The computation of dominators is a central tool in program optimization and code generation, and it has applications in other diverse areas including constraint programming, circuit testing, and biology. In this paper we survey recent results, applications, and open problems related to the notion of dominators in directed graphs, including dominator verification and certification, computing independent spanning trees, and connectivity and path-determination problems in directed graphs.**

I. Introduction

A *flow graph* $G = (V, A, s)$ is a directed graph with a distinguished *start* vertex s such that every vertex is reachable from s. A vertex u *dominates* a vertex v (u is a *dominator* of v) if every path from s to v contains u; if $u \neq v$, u is a *strict dominator* of v. The dominator relation is transitive and can be represented in compact form as a tree D, the *dominator tree* of G. Tree D is rooted at s and is such that every vertex dominates all its descendants and is dominated by all its ancestors. See Figure 1. The parent $d(v)$ of v in D is the *immediate dominator* of v, the unique strict dominator of v dominated by all strict dominators of v. Tree D is *flat* if each vertex $v \neq s$ has $d(v) = s$. Let T be a rooted tree with vertex set V. Tree T has the *parent property* if for all $(v, w) \in A$, the parent of w in T is an ancestor of v in T. Tree T has the *sibling property* if v does not dominate w for all siblings v and w. The parent and sibling properties are necessary and sufficient for a tree to be the dominator tree [1], [2]. A flow graph $G = (V, A, s)$ is *reducible* if every strongly connected subgraph S has a single entry vertex v such every path from s to a vertex in S contains v [3], [4]. A reducible flow graph becomes acyclic when every arc (v, w) such that w dominates v is deleted [4].

The dominator tree is a central tool in program optimization and code generation [5]. Compilers use dominators extensively during program analysis and optimization, for various goals such as natural loop detection (which enables a host of optimizations), structural analysis [6], scheduling [7], and the computation of dependence graphs and static single-assignment forms [8]. Dominators have applications in other diverse areas including constraint programming [9], circuit testing [10], theoretical biology [11], memory profiling [12], connectivity and path-determination problems [13]–[15], and the analysis of diffusion networks [16].

II. Algorithms for Computing Dominators

The problem of finding dominators has been extensively studied. In 1972 Allen and Cocke [17] showed that the dominator relation can be computed iteratively from a set of data-flow equations. A direct implementation of this method has an $O(mn^2)$ worst-case time bound. Cooper, Harvey, and Kennedy [18] presented a clever tree-based space-efficient implementation of the iterative algorithm. Although it does not improve the $O(mn^2)$ worst-case time bound, the tree-based version is much more efficient in practice. Purdom and Moore [19] introduced a simple, reachability-based algorithm with time complexity $O(mn)$. Improving on previous work by Tarjan [20], Lengauer and Tarjan [21] gave two near-linear-time algorithms for computing D that run fast in practice and have been used in many applications. The simpler of these runs in $O(m \log_{(m/n+1)} n)$ time, and the other runs in $O(m\alpha(m, n))$ time, where α is a functional inverse of Ackermann's function [22]. (Note that $m \geq n - 1$ by the assumption that all vertices are reachable from s.) The core of the computations performed by the Lengauer-Tarjan algorithm is to find minima of a function defined on the paths of a depth-first search spanning tree of G [23]. This can be done efficiently by a data structure that supports *link* and *eval* operations, which resemble the *unite* and *find* operations of a disjoint set union data structure [22], but involve a more elaborate use of path compression and tree balancing. With a simple linking strategy the algorithm runs in $O(m \log_{(m/n+1)} n)$ time, and with a more complicated balanced linking strategy it achieves the $O(m\alpha(m, n))$ time bound. Subsequently, more-complicated but truly linear-time algorithms to compute dominators were discovered [24]–[27]; these algorithms are based on the Lengauer-Tarjan algorithm and achieve linear time by incorporating several other techniques, including the precomputation of answers to small subproblems. Very recently, Gabow [28] and Fraczak et al. [29] presented linear-time algorithms that are based on a different approach, and require only simple data structures and a data structure for static tree set union [30]. Gabow's algorithm uses the concept of minimal-set posets [31], [32], while the algorithm of Fraczak et al. uses vertex contractions. Of those linear-time algorithms, [24], [26], [28], [29] use bit-manipulation techniques, so they run on the random-access-machine (RAM) model of computation. On the other hand, [25], [27] are implementable on the less powerful pointer-machine model [33]. Ramalingam and Reps [34] presented an incremental algorithm for finding dominators in an acyclic graph. Their algorithm uses a data structure that

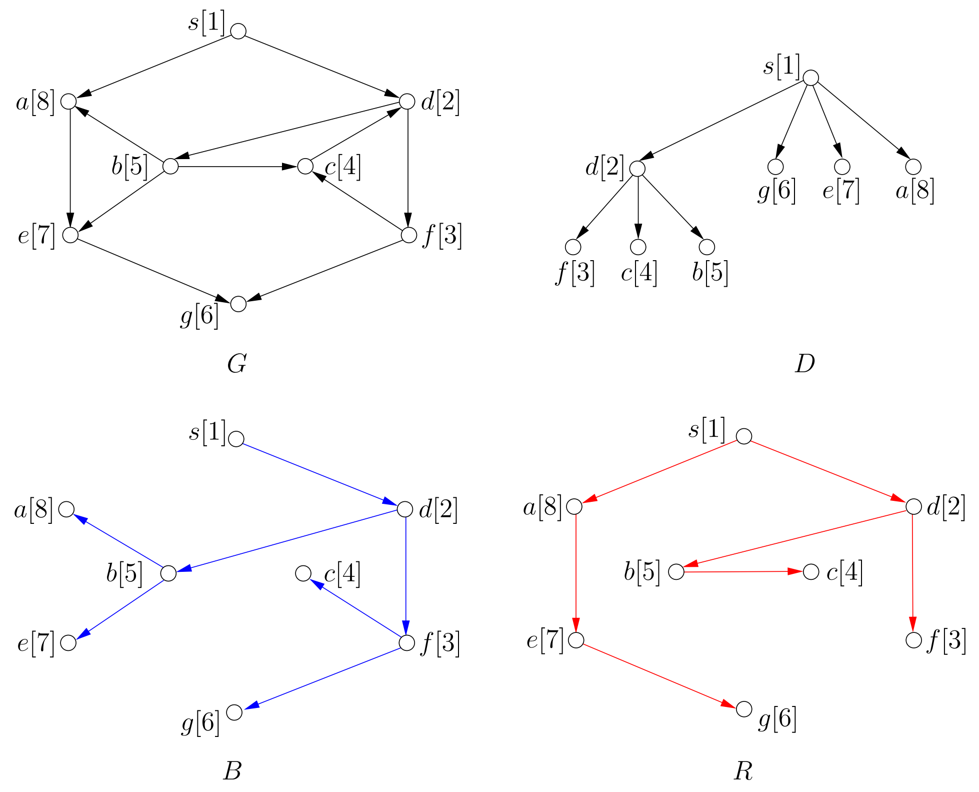

Fig. 1. A flow graph G, its dominator tree D with vertices numbered in low-high order (numbers in brackets), and two strongly independent spanning trees B and R.

computes nearest common ancestors in a tree that grows by leaf additions. (See also [35], [36].) For this incremental nearest common ancestors problem, Gabow [37] gave an $O(m)$-time RAM algorithm, and Alstrup and Thorup [38] gave an $O(m \log \log n)$-time pointer machine algorithm. These results give implementations of the Ramalingam-Reps algorithm that run in $O(m)$ time on a RAM and in $O(m \log \log n)$ time on a pointer machine. Ramalingam [36] showed how to reduce the problem of computing dominators in an arbitrary flow graph to computing dominators in an acyclic graph. His reduction uses static-tree disjoint set union, so it runs in $O(m\alpha(m,n))$ time on a pointer machine [22], and in $O(m)$ time on a RAM [30]. Therefore, the combination of any linear-time algorithm for computing dominators in an acyclic graph with Ramalingam's reduction gives a linear-time RAM algorithm for computing dominators in a general graph.

A hybrid algorithm, named SNCA, which combines the simple version of the Lengauer-Tarjan algorithm with the Cooper-Harvey-Kennedy algorithm, was presented in [39]. This algorithm runs in $O(n^2)$ time in the worst-case but is reported to perform much better in practice. An experimental study of algorithms for computing dominators was presented in [39], where careful implementations of both versions of the Lengauer-Tarjan algorithm, the iterative algorithm of Cooper, Harvey and Kennedy, and SNCA were given. In these experimental results the performance of all these algorithms was similar, but the simple version of the Lengauer-Tarjan algorithm and SNCA were most consistently fast, and their advantage increased as the input graph got bigger or more complicated. The graphs used in [39] were taken from the application areas in program optimization, circuit testing, and biology, and have moderate size (at most a few thousand vertices and edges) and simple enough structure that they can be efficiently processed by the iterative algorithm. Recent experimental results for computing dominators in large graphs are reported in [14], [40], [41]. There it is apparent that the simple iterative algorithms are not competitive with the more sophisticated algorithms based on Lengauer-Tarjan. The graphs used in these experiments were taken from applications of dominators in memory profiling [12], [42], testing 2-vertex connectivity and computing sparse 2-vertex connected spanning subgraphs [13], [14], and computing strong articulation points and strong bridges in directed graphs [15], which typically involve much larger and complicated graphs.

III. Dominator Verification and Certification

The most efficient algorithms for computing dominators that we mentioned in Section II have some conceptual complexities. This gives rise to the question of how does one know that the output produced by these fast but complicated algorithms for finding dominators is correct. This is an important instance of software verification, a prominent and costly problem in software engineering [43]. One approach to deal with this problem is to formally prove the correctness of the program, but this may currently be beyond reach for programs that implement sophisticated algorithms. Another option is to implement a *checker* [44], which given an input-output pair verifies that this pair satisfies the desired input-output relation. A third option is to provide a *certifying algorithm* [43], i.e., an algorithm that produces not only the desired output but also a *correctness certificate* that can be validated by a *certifier*. A certifier is an algorithm that is given an input-output pair and a certificate, and uses the certificate to verify that the input-output pair satisfies the desired input-output relation. The certified algorithm should be as efficient (to within a constant factor) as the most efficient non-certifying algorithm. Moreover the certifier should be much simpler and easy to prove correct.

In the dominator verification problem, running time is not the only measure of simplicity, since $O(m)$ time is necessary and sufficient for constructing the dominator tree, and hence for verifying it. We are not aware of any formal verification of a fast algorithm to compute dominators; a much simpler but much slower algorithm has been formally verified [45]. The problem of verifying a dominator tree, i.e., providing a checker that, given a flow graph G and a tree T, verifies that T is the dominator tree of G, was considered in [1], [46]. There, linear-time algorithms were given, which are based on the concepts of *headers* and *loop nesting forests* [36], [47]. Although these algorithms are simpler than computing dominators from scratch, they are not straightforward. A more satisfying answer to the dominator verification problem was achieved in [1], [2]. There, it is shown that an appropriate certificate for a dominator tree D of a flow graph G is a preorder of the vertices of D with a certain property, which is called *low-high*. A preorder is of D is low-high on G if, for all $v \neq s$, $(d(v), v) \in A$ or there are two arcs $(u, v) \in A$, $(w, v) \in A$ such that u is less than v, v is less than w, and w is not a descendant of v. Figure 1 gives an example. Verifying that an order is low-high is entirely straightforward and can be done easily in linear time. Also, computing a low-high order given G and D can be done in linear-time. Then, by augmenting an efficient algorithm to compute D so that it also computes a low-high order, one obtains an efficient certifying algorithm.

Efficient implementations of certified algorithms for computing dominators, based on the Lengauer-Tarjan algorithm, were presented in [41]. There it is reported that the computation of the certificate (low-high order) adds a small overhead, usually within a factor of two, to the running time of the non-certified algorithm.

IV. Independent Spanning Trees

Consider a flow graph $G = (V, A, s)$ and two spanning trees B and R of G, rooted at s. We call such spanning trees *disjoint*, if, for any v, the paths from s to v in B and R share only s and v. We call B and R *strongly disjoint*, if, for any distinct vertices v and w, either the path in B from s to v and the path in R from s to w share only s, or the path in R from s to v and the path in B from s to w share only s. Whitty [48] proved that any flow graph G with *flat* dominator tree has two strongly disjoint spanning trees. Plehn [49] and independently Cheriyan and Reif [50] gave simpler proofs of Whitty's result using what Cheriyan and Reif called a directed st-numbering as an intermediary. A *directed st-numbering* of a directed graph $G = (V, A)$ with two distinct vertices s and t, is a numbering $\pi : V \mapsto \{1, 2, \dots, n\}$ of its vertices such that $\pi(s) = 1$, $\pi(t) = n$, and every other vertex v has entering arcs from a vertex u and a vertex w with $\pi(u) < \pi(v) < \pi(w)$. The proofs in [48]–[50] imply polynomial-time constructions of directed st-numberings and of strongly disjoint spanning trees, which seem to require $\Omega(nm)$ time in the worst case. Later, Huck [51] gave an $O(nm)$-time algorithm to find two disjoint spanning trees.

The above concepts were extended to arbitrary flow graphs in [1], [2], [46]. Let G be a flow graph with vertex set V, arc set A, and start vertex s. Two spanning trees B and R rooted at s are *independent* if for all v, the paths from s to v in B and R share only the dominators of v; B and R are *strongly independent* if for every pair of vertices v and w, either the path in B from s to v and the path in R from s to w share only the common dominators of v and w, or the path in R from s to v and the path in B from s to w share only the common dominators of v and w. These definitions, also illustrated in Figure 1, extend the notions of disjoint and strongly disjoint spanning trees, respectively; low-high orders extend the notion of directed st-numberings. Moreover, [1], [2], [46] presented linear-time algorithms for their construction.

Independent spanning trees are closely related to low-high orders, i.e., the certificate for dominator trees mentioned in Section III: Given two independent spanning trees of G one can construct in linear time a low-high order; conversely, given a low-high order of G, one can construct two strongly independent spanning trees in linear time [1], [2], [46].

V. Connectivity and Disjoint-Paths Problems

A directed (undirected) graph is *k-vertex connected* if it has at least $k+1$ vertices and the removal of any set of at most $k-1$ vertices leaves the graph strongly connected (connected). The problem of testing the 2-vertex connectivity of a directed graph can be reduced in linear time to verifying that the dominator tree of a flow graph is flat [13], [15]. This way we obtain a linear-time algorithm to test 2-vertex connectivity, either by using a linear-time algorithm to compute dominators, or a linear-time algorithm to verify that the dominator tree of given flow graph is flat.

The linear-time algorithms [1], [2], [46] for computing two (strongly) independent spanning trees can be used in a data structure that computes pairs of vertex-disjoint s-t paths in 2-vertex connected directed graphs (for any two query vertices s and t) [13], and in fast algorithms for approximating the smallest 2-vertex connected spanning subgraph of a directed graph [14].

In [52], Tholey considers the following disjoint-paths problem: Let (u_i, v_i), $1 \leq i \leq k$, be k pairs of vertices of a directed acyclic graph with two distinguished start vertices s_1 and s_2. For each pair (u_i, v_i), we wish to test if there are two vertex disjoint paths $P_1 = (s_1, \dots, t_1)$ and $P_2 = (s_2, \dots, t_2)$, where $\{t_1, t_2\} = \{u_i, v_i\}$, and to construct such paths if they exist. Tholey showed how to test the existence of P_1 and P_2 in constant time and how to produce them in $O(|P_1| + |P_2|)$ time after linear-time preprocessing. He then uses this result to give a linear-time algorithm for the 2-disjoint paths problem on a directed acyclic graph. Tholey's algorithm for testing the existence of P_1 and P_2 and for constructing them uses dominator trees, shortest-path trees, a topological order of the directed acyclic graph, and other structures. The use of a low-high order gives us an alternative solution that works for a general directed graph G [2].

VI. Dynamic Dominators

Now we consider the problem of dynamically maintaining the dominator relation of a flow graph that undergoes both insertions and deletions of edges. Vertex insertions and deletions can be simulated using combinations of edge updates. A graph problem is *fully dynamic* if it requires to process both insertions and deletions of edges, *incremental* if it requires to process edge insertions only and *decremental* if it requires to process edge deletions only. The fully dynamic dominators problem arises in various applications, such as data flow analysis and compilation [53]. Moreover, [13], [15] imply that a fully dynamic dominators algorithm can be used for dynamically testing 2-vertex connectivity, and maintaining the strong articulation points and strong bridges of a directed graph. The decremental dominators problem appears in the computation of 2-connected components in directed graphs [15].

Consider the effect that a single edge update (insertion or deletion) has on the dominator tree D. Let (x, y) be the inserted or deleted edge. We say that a vertex v is *affected* by the update if its immediate dominator changes. It is easy to verify that an edge insertion can violate the parent property of D, while an edge deletion can violate the sibling property of D. The difficulty in updating the dominance relation lies on two facts: (i) An affected vertex can be arbitrarily far from the updated edge, and (ii) a single update may affect many vertices. In fact, we can construct sequences of $\Theta(n)$ edge insertions (deletions) such that each single insertion (deletion) affects $\Theta(n)$ vertices [54]. This implies a lower bound of $\Omega(n^2)$ time for any algorithm that maintains D explicitly through a sequence of $\Omega(n)$ edge insertions or a sequence of $\Omega(n)$ edge deletions, and a lower bound of $\Omega(mn)$ time for any algorithm that maintains D through an intermixed sequence of $\Omega(m)$ edge insertions and deletions.

The problem of updating the dominator relation has been studied in [34], [53]–[58]. However, a worst-case complexity bound for a single update better than $O(m)$ has been only achieved for special cases, mainly for incremental or decremental problems. Specifically, the algorithm of Cicerone et al. [53] achieves $O(n \max\{k, m\} + q)$ running time for processing a sequence of k edge insertions interspersed with q queries of the type "does x dominate y?", for a flow graph with n vertices and initially m edges. The same bound is also achieved for a sequence of k deletions, but only for a reducible flow graph. This algorithm does not maintain the dominator relation in a tree but in an $n \times n$ matrix, so a query can be answered in constant time. Alstrup and Lauridsen describe in a technical report [55] an algorithm that maintains the dominator tree through a sequence of k edge insertions interspersed with q queries in $O(m \min\{k, n\} + q)$ time. In this bound m is the number of edges after all insertions. However, the description of this algorithm is incomplete and seem to contain some incorrect arguments [54]. Recently, [54] presented an algorithm that uses a depth-based search of the dominator tree in order to locate the affected vertices, using some ideas from [55], [58]. The algorithm is fully dynamic and requires $O(m)$ time for insertion and $O(n^2)$ time for deletion, but is reported to perform much better in practice [54]. For the incremental problem, the algorithm of [54] maintains the dominator tree through a sequence of k edge insertions in $O(m \min\{k, n\} + kn)$ time, where m is the number of edges after all insertions.

VII. Further Applications and Open Problems

We conclude by mentioning some further applications of our constructions and some open problems.

a) Low-high orders: In [2], [41] it is conjectured that a low-high order is a topological order of a directed graph induced by the arcs of the independent spanning trees B and R constructed by the algorithms in [2], [46]. Experiments on large graphs, reported in [41] suggest that this is true, but there is no proof.

Another related problem is whether there is a simple way to extend the iterative algorithm of Cooper, Harvey and Kennedy [18], or incremental algorithms for computing dominators [34], [54], [58] so that they also compute a low-high order of the dominator tree. Such an extension would make these into certifying algorithms.

b) Independent Spanning Trees: Verifying that two spanning trees are independent or strongly disjoint is not easy. Given a flow graph G, its dominator tree D, and two trees B and R, we can test in $O(n \log n)$ time if B and R are independent spanning trees of G using the techniques of [59]. Is there a linear-time algorithm for this test? Is there a fast way to test if B and R are strongly independent?

Huck [60] showed that for any $k \geq 3$ there is a k-connected graph that does not have k independent spanning trees. This result does not hold in special cases, such as planar graphs [61], and acyclic graphs [62], [63]. For undirected graphs, Itai and Rodeh [64], [65] 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 [66], who also gave a corresponding $O(n^2)$-time algorithm, and by Itai and Zehavi [67]. Curran, Lee and Yu [68] provided a $O(n^3)$-time algorithm that constructs four independent spanning trees of a 4-connected graph, thus proving the $k = 4$ case. To the best of our knowledge, the case $k \geq 5$ is open.

c) Interprocedural Dominators: The most efficient algorithms to compute dominators operate in the *intraprocedural* setting where all flow graph paths are valid. This suffices for standard compilers, since they compute dominators in the flow graph of each procedure separately. In the context of whole-program analysis, however, we deal with an *interprocedural* flow graph that is composed from all the procedures in the program. An interprocedural flow graph contains *context-sensitive* edges, such that every valid path in the flow graph must have a proper nesting of procedure calls and returns. As a result, the transitive reduction D of the interprocedural dominator relation is a directed acyclic graph (DAG), and cannot be computed by the most efficient algorithms for finding intraprocedural dominators. In practice interprocedural flow graphs can be very large, and therefore the use of algorithms that are not time and space efficient is not a viable option in many cases. Computing interprocedural dominators efficiently is an important step towards whole-program optimization [69]. Reps, Horwitz and Sagiv [70] gave polynomial-time algorithms for a general class of interprocedural dataflow analysis problems by transforming them to a special kind of graph reachability. A reachability-based algorithm for interprocedural dominance was presented by Ezick, Bilardi and Pingali [71]. Their algorithm builds the complete dominance relation in $O(mn)$ time and $O(n^2)$ space. An iterative algorithm was given by de Sutter, van Put and de Bosschere [69]. Although its running time is $O(mn^3)$ in the worst case it is reported to perform well in practice with the help of several heuristics. The important open question here is whether there is a provably fast and practical algorithm to compute interprocedural dominators.

References

[1] L. Georgiadis and R. E. Tarjan, "Dominators, directed bipolar orders, and independent spanning trees," in *Proc. 39th Int'l. Coll. on Automata, Languages, and Programming*, 2012, pp. 375–386.

[2] ——, "Dominator tree certification and independent spanning trees," *CoRR*, vol. abs/1210.8303, 2012.

[3] M. S. Hecht and J. D. Ullman, "Characterizations of reducible flow graphs," *Journal of the ACM*, vol. 21, no. 3, pp. 367–375, 1974.

[4] R. E. Tarjan, "Testing flow graph reducibility," *J. Comput. Syst. Sci.*, vol. 9, no. 3, pp. 355–365, 1974.

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

[6] M. Sharir, "Structural analysis: A new approach to flow analysis in optimizing compilers," *Computer Languages*, vol. 5, no. 3, pp. 141–153, 1980.

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

[8] 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*, vol. 13, no. 4, pp. 451–490, 1991.

[9] L. Quesada, P. Van Roy, Y. Deville, and R. Collet, "Using dominators for solving constrained path problems," in *Proc. 8th International Conference on Practical Aspects of Declarative Languages*, 2006, pp. 73–87.

[10] 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.

[11] S. Allesina and A. Bodini, "Who dominates whom in the ecosystem? Energy flow bottlenecks and cascading extinctions," *Journal of Theoretical Biology*, vol. 230, no. 3, pp. 351–358, 2004.

[12] E. K. Maxwell, G. Back, and N. Ramakrishnan, "Diagnosing memory leaks using graph mining on heap dumps," in *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, ser. KDD '10, 2010, pp. 115–124.

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

[14] ——, "Approximating the smallest 2-vertex connected spanning subgraph of a directed graph," in *Proc. 19th European Symposium on Algorithms*, 2011, pp. 13–24.

[15] G. F. Italiano, L. Laura, and F. Santaroni, "Finding strong bridges and strong articulation points in linear time," *Theoretical Computer Science*, vol. 447, no. 0, pp. 74–84, 2012. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0304397511009303

[16] M. Gomez-Rodriguez and B. Schölkopf, "Influence maximization in continuous time diffusion networks," in *29th International Conference on Machine Learning (ICML)*, 2012.

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

[18] K. D. Cooper, T. J. Harvey, and K. Kennedy, "A simple, fast dominance algorithm," Rice Computer Science, Tech. Rep. TR-06-38870, 2006.

[19] P. W. Purdom, Jr. and E. F. Moore, "Algorithm 430: Immediate predominators in a directed graph," *Communications of the ACM*, vol. 15, no. 8, pp. 777–778, 1972.

[20] R. E. Tarjan, "Finding dominators in directed graphs," *SIAM Journal on Computing*, vol. 3, no. 1, pp. 62–89, 1974.

[21] T. Lengauer and R. E. Tarjan, "A fast algorithm for finding dominators in a flowgraph," *ACM Transactions on Programming Languages and Systems*, vol. 1, no. 1, pp. 121–41, 1979.

[22] R. E. Tarjan, "Efficiency of a good but not linear set union algorithm," *Journal of the ACM*, vol. 22, no. 2, pp. 215–225, 1975.

[23] ——, "Applications of path compression on balanced trees," *Journal of the ACM*, vol. 26, no. 4, pp. 690–715, 1979.

[24] S. Alstrup, D. Harel, P. W. Lauridsen, and M. Thorup, "Dominators in linear time," *SIAM Journal on Computing*, vol. 28, no. 6, pp. 2117–32, 1999.

[25] A. L. Buchsbaum, L. Georgiadis, H. Kaplan, A. Rogers, R. E. Tarjan, and J. R. Westbrook, "Linear-time algorithms for dominators and other path-evaluation problems," *SIAM Journal on Computing*, vol. 38, no. 4, pp. 1533–1573, 2008.

[26] 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*, vol. 20, no. 6, pp. 1265–96, 1998, corrigendum in 27(3):383-7, 2005.

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

[28] H. N. Gabow, "A poset approach to dominator computation," 2013, unpublished manuscript 2010, revised unpublished manuscript.

[29] W. Fraczak, L. Georgiadis, A. Miller, and R. E. Tarjan, "Finding dominators via disjoint set union," *Journal of Discrete Algorithms*, vol. 23, no. 0, pp. 2–20, 2013.

[30] 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*, vol. 30, no. 2, pp. 209–21, 1985.

[31] H. N. Gabow, "Applications of a poset representation to edge connectivity and graph rigidity," in *Proc. 32th IEEE Symp. on Foundations of Computer Science*, 1991, pp. 812–821.

[32] ——, "The minimal-set poset for edge connectivity," 2013, unpublished manuscript.

[33] R. E. Tarjan, "A class of algorithms which require nonlinear time to maintain disjoint sets," *Journal of Computer and System Sciences*, vol. 18, no. 2, pp. 110–27, 1979.

[34] G. Ramalingam and T. Reps, “An incremental algorithm for maintaining the dominator tree of a reducible flowgraph,” in *Proc. 21th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, 1994, pp. 287–296.

[35] S. Alstrup and P. W. Lauridsen, “A simple and optimal algorithm for finding immediate dominators in reducible graphs,” Dept. of Computer Science, U. Copenhagen, Tech. Rep. DIKU TOPPS D-261, 1996.

[36] G. Ramalingam, “On loops, dominators, and dominance frontiers,” *ACM Transactions on Programming Languages and Systems*, vol. 24, no. 5, pp. 455–490, 2002.

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

[38] S. Alstrup and M. Thorup, “Optimal pointer algorithms for finding nearest common ancestors in dynamic trees,” *Journal of Algorithms*, vol. 35, pp. 169–88, 2000.

[39] L. Georgiadis, R. E. Tarjan, and R. F. Werneck, “Finding dominators in practice,” *Journal of Graph Algorithms and Applications (JGAA)*, vol. 10, no. 1, pp. 69–94, 2006.

[40] D. Firmani, G. F. Italiano, L. Laura, A. Orlandi, and F. Santaroni, “Computing strong articulation points and strong bridges in large scale graphs,” in *Proc. 10th Int’l. Symp. on Experimental Algorithms*, 2012, pp. 195–207.

[41] L. Georgiadis, L. Laura, N. Parotsidis, and R. E. Tarjan, “Dominator certification and independent spanning trees: An experimental study,” in *Proc. 12th Int’l. Symp. on Experimental Algorithms*, 2013, pp. 284–295.

[42] N. Mitchell, “The runtime structure of object ownership,” in *20th European Conference on Object-Oriented Programming*, 2006, pp. 74–98.

[43] R. M. McConnell, K. Mehlhorn, S. Näher, and P. Schweitzer, “Certifying algorithms,” *Computer Science Review*, vol. 5, no. 2, pp. 119–161, 2011.

[44] M. Blum and S. Kannan, “Designing programs that check their work,” *Journal of the ACM*, vol. 42, no. 1, pp. 269–291, 1995.

[45] J. Zhao and S. Zdancewic, “Mechanized verification of computing dominators for formalizing compilers,” in *Proc. 2nd International Conference on Certified Programs and Proofs*. Springer, 2012, pp. 27–42.

[46] L. Georgiadis and R. E. Tarjan, “Dominator tree verification and vertex-disjoint paths,” in *Proc. 16th ACM-SIAM Symp. on Discrete Algorithms*, 2005, pp. 433–442.

[47] R. E. Tarjan, “Edge-disjoint spanning trees and depth-first search,” *Acta Informatica*, vol. 6, no. 2, pp. 171–85, 1976.

[48] R. W. Whitty, “Vertex-disjoint paths and edge-disjoint branchings in directed graphs,” *Journal of Graph Theory*, vol. 11, pp. 349–358, 1987.

[49] J. Plehn, “Über die existenz und das finden von subgraphen,” Ph.D. dissertation, University of Bonn, Germany, May 1991.

[50] J. Cheriyan and J. H. Reif, “Directed s-t numberings, rubber bands, and testing digraph k-vertex connectivity,” *Combinatorica*, pp. 435–451, 1994, also in SODA ’92.

[51] A. Huck, “Independent trees in graphs,” *Graphs and Combinatorics*, vol. 10, pp. 29–45, 1994.

[52] T. Tholey, “Linear time algorithms for two disjoint paths problems on directed acyclic graphs,” *Theoretical Computer Science*, 2012, in press.

[53] S. Cicerone, D. Frigioni, U. Nanni, and F. Pugliese, “A uniform approach to semi-dynamic problems on digraphs,” *Theor. Comput. Sci.*, vol. 203, pp. 69–90, August 1998.

[54] L. Georgiadis, G. F. Italiano, L. Laura, and F. Santaroni, “An experimental study of dynamic dominators,” in *Proc. 20th European Symposium on Algorithms*, 2012, pp. 491–502.

[55] S. Alstrup and P. W. Lauridsen, “A simple dynamic algorithm for maintaining a dominator tree,” Department of Computer Science, University of Copenhagen, Tech. Rep. 96-3, 1996.

[56] M. D. Carroll and B. G. Ryder, “Incremental data flow analysis via dominator and attribute update,” in *Proc. 15th ACM POPL*, 1988, pp. 274–284.

[57] K. Patakakis, L. Georgiadis, and V. A. Tatsis, “Dynamic dominators in practice,” in *Proc. 16th Panhellenic Conference on Informatics*, 2011, pp. 100–104.

[58] V. C. Sreedhar, G. R. Gao, and Y. Lee, “Incremental computation of dominator trees,” *ACM Transactions on Programming Languages and Systems*, vol. 19, pp. 239–252, 1997.

[59] L. Georgiadis, S. D. Nikolopoulos, and L. Palios, “Join-reachability problems in directed graphs,” *Theory of Computing Systems*, pp. 1–33, 2013.

[60] A. Huck, “Disproof of a conjecture about independent branchings in k-connected directed graphs,” *Journal of Graph Theory*, vol. 20, no. 2, pp. 235–239, 1995.

[61] ——, “Independent trees and branchings in planar multigraphs,” *Graphs and Combinatorics*, vol. 15, pp. 211–220, 1999.

[62] ——, “Independent branchings in acyclic digraphs,” *Discrete Math*, vol. 199, pp. 245–249, 1999.

[63] F. S. Annexstein, K. A. Berman, T. Hsu, and R. P. Swaminathan, “A multi-tree routing scheme using acyclic orientations,” *Theor. Comput. Sci.*, vol. 240, no. 2, pp. 487–494, 2000.

[64] A. Itai and M. Rodeh, “Three tree-paths,” in *Proc. 25th IEEE Symp. on Foundations of Computer Science*, 1984, pp. 137–147.

[65] ——, “The multi-tree approach to reliability in distributed networks,” *Information and Computation*, vol. 79, no. 1, pp. 43–59, 1988.

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

[67] A. Itai and A. Zehavi, “Three tree-paths,” *Journal of Graph Theory*, vol. 13, pp. 175–188, 1989.

[68] S. Curran, O. Lee, and X. Yu, “Finding four independent trees,” *SIAM Journal on Computing*, vol. 35, no. 5, pp. 507–537, 2006.

[69] B. de Sutter, L. van Put, and K. de Bosschere, “A practical interprocedural dominance algorithm,” *ACM Trans. Program. Lang. Syst.*, vol. 29, no. 4, 2007.

[70] 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*, June 1995, pp. 49–61.

[71] J. Ezick, G. Bilardi, and K. Pingali, “Efficient computation of interprocedural control dependence,” Cornell Computer Science Department, Technical Report TR2001-1850, 2001.