Linear-Time Pointer-Machine Algorithms for Least Common Ancestors, MST Verification, and Dominators

Adam L. Buchsbaum Haim Kaplan Anne Rogers
Jeffery R. Westbrook
AT&T Labs, 180 Park Ave., Florham Park, NJ 07932, USA
{alb,hkl,amr,jeffw}@research.att.com

Abstract

We present two new data structure tools—disjoint set union with bottom-up linking, and pointer-based radix sort—and combine them with bottom-level microtrees to devise the first linear-time pointer-machine algorithms for off-line least common ancestors, minimum spanning tree (MST) verification, randomized MST construction, and computing dominators in a flowgraph.

1 Introduction

We study four problems—off-line least common ancestors (LCAs), minimum spanning tree (MST) verification, MST construction, and computing dominators in a flowgraph—for which gaps exist between the best known pointer-machine and RAM algorithms, as summarized by Table 1. We present the first linear-time pointer-machine algorithms for these problems, solving several outstanding open questions [9, 17, 18]. In the MST construction case, the time bound is expected. Additionally, our algorithms are simpler than their RAM counterparts.

A pointer machine [28] allows binary comparisons between data, arithmetic operations on data, dereferencing of pointers, and equality tests on pointers. It does not permit pointer arithmetic or tests other than equality on pointers and thus is less powerful than the RAM model. Pointer machines are powerful enough, however, to simulate functional programming languages such as LISP and ML.

For LCAs, MST verification, and dominators, the previous pointer-machine algorithms use disjoint set union (DSU), yielding the inverse Ackermann α terms. The corresponding RAM algorithms exploit the sensitivity of α to the graph density by partitioning the graphs into small subpieces, called *microtrees*, solving the problem on the microtrees by precomputation and table lookup, and running the standard pointer-based algorithm on a smaller, induced graph, which becomes dense enough for α to become constant. (If $m/n = \Omega(\log^{(O(1))} n)$, $\alpha(m,n) = O(1)$.) The table lookups require a RAM. The linear-time RAM MST construction algorithms [13, 17] do not use this framework, but the only component of the Karger, Klein, and Tarjan randomized MST algorithm [17] that up to now required a RAM is an MST verification subroutine.

Each of the above algorithms is based on a traversal of a spanning tree or arborescence, T. The traversal induces a sequence of unions of the form $(v, \mathit{parent}(v))$ for vertices $v \in T$. The algorithms restrict the unions to occur "bottom up:" no union $(v, \mathit{parent}(v))$ is performed until all unions have been performed in v's subtree. Assuming this restriction, we devise a simple modification to the standard pointer-based DSU data structure, which reduces the $\alpha(m,n)$ term to $\alpha(m,l)$, where l is the number of leaves in the union tree. Thus, under bottom-up linking, we need not restrict the global graph density m/n to make $\alpha(m,n)$ constant. We need only restrict the number of leaves l appropriately. We exploit this result by partitioning T into microtrees only at the bottom, thereby restricting the number of leaves in the rest of T. This approach simplifies previous microtree techniques, which partition all of T into microtrees.

The linear-time RAM algorithms use random access to build and index a table that contains results of relevant computations on all possible distinct microtrees. Each microtree in the input is encoded as an integer, and a constant-time table lookup finds its solution. We eliminate the use of random access by identifying duplicate microtrees, computing on each distinct microtree once, and copying the results to the duplicates. We introduce a pointer-based radix sort to organize microtree encodings, which lets us process the microtrees in linear time on a pointer machine.

We describe our techniques in the context of the LCA problem, which we define in Section 2. Section 3 gives our DSU result, motivating the restriction of microtrees to the bottom of a tree. Section 4 gives our pointer-based radix sort technique for processing microtrees on a pointer machine, completing our new LCA algorithm. Sections 5 and 6 apply all three techniques—DSU with bottom-up linking, bottom-

To appear in the *Thirtieth ACM Symp. on the Theory of Computing*, 1998.

Table 1: Time bounds. n is the number of vertices, m is the number of edges/arcs, and p is the number of LCA queries. $\alpha(m, n)$ is the standard functional inverse of the Ackermann function.

Problem	Previous Pointer-Machine Bound	Previous RAM Bound
Off-line LCAs	$O(p\alpha(p, n) + n)$ [1]	$O(n + p)$ [16,25]
MST Verification	$O(m\alpha(m, n) + n)$ [27]	$O(n + m)$ [9,18]
MST Construction	$O(m\alpha(m, n) \log \alpha(m, n) + n)$ [7]	$O(n + m)$ [13,17]
Dominators	$O(m\alpha(m, n) + n)$ [20]	$O(n + m)$ [3,15]

level microtrees, and pointer-based radix sort—to the MST verification (and construction) and dominators problems.

2 Least Common Ancestors

Let $T = (V, E)$ be a tree with root r, and let $P \subseteq V \times V$ be a set of pairs of vertices of T. We wish to compute the least common ancestor *lca*(x, y) for each pair $\{x, y\} \in P$; this is the *off-line least common ancestors (LCAs) problem.* We assume that T is given in adjacency list format and that associated with each $v \in V$ is a list of pairs, P_v, in P that contain v [27].

The best known pointer-machine algorithm for off-line LCAs, given by Aho, Hopcroft, and Ullman (AHU) [1], runs in $O(p\alpha(p, n) + n)$ time, where $n = |V|$ and $p = |P|$. Gabow and Tarjan's linear-time DSU result [14] can make the AHU LCA algorithm run in linear time on a RAM. Harel and Tarjan [16] and Schieber and Vishkin [25] solve the *on-line LCAs problem*, in which P is not given a priori, in $O(n + p)$ time on a RAM.

The AHU algorithm uses DSU as follows. At any time during the execution of the algorithm, each set corresponds to a subtree of T. The name of a set is the root of the corresponding subtree. Initially, each vertex comprises a singleton set. The algorithm performs a depth-first search (DFS) of T. Recall a vertex is said to be *scanned* if it has been completely processed by the DFS. When the DFS backtracks through a vertex v, for every pair $\{v, w\} \in P_v$ such that w has already been scanned, *lca*(v, w) is assigned to be the result of *find*(w). After processing P_v, we perform *union*$(v, p(v))$, where $p(v)$ is the parent of v in T. Correctness of this algorithm follows from basic properties of DFS. Implementing the DSU data structure using balanced linking and path compression [29] gives the stated time bound.

Notice that the unions occur "bottom up" due to the postorder processing: *union*$(v, p(v))$ occurs only after *union*(x, v) occurs for all children x of v. In the next section, we analyze the behavior of DSU under such a restriction. The analysis motivates us to restrict the number of leaves in T by removing small subtrees from the bottom of T, which allows the DSU data structure to run in linear time. This restriction also simplifies the application of Gabow and Tarjan's technique [14] to the AHU algorithm.

3 Disjoint Set Union with Bottom-Up Linking

Let U be a set of n vertices, each initially identified with a singleton set. The sets are subject to the standard DSU operations: *union*(A, B, C), which unites sets A and B and names the result C, and *find*(u), which returns the name of the set containing u. Using the standard forest data structure with union-by-size and path compression, $n - 1$ unions intermixed with m finds take $O(m\alpha(m, n) + n)$ time [29].

We can improve this time bound, given sufficient restrictions on the order of the unions. Previously, Gabow and Tarjan [14] used a priori knowledge of the unordered set of unions to implement the union and find operations in $O(m+n)$ time. We do not require advance knowledge of the unions, only that their order be constrained. Other results on improved bounds for path compression [6, 21, 23] generally restrict the order in which finds, not unions, are performed.

The following theorem shows that requiring the unions to "favor" a small set of vertices results in a linear time bound. Designate l vertices to be *special* and the remaining $n - l$ to be *ordinary*.

Theorem 3.1 *Consider n vertices such that l are special and the remaining $n - l$ are ordinary. Let σ be a sequence of $n - 1$ unions and m finds such that each union involves at least one set that contains at least one special vertex. Then the operations can be performed in $O(m\alpha(m, l) + n)$ time.*

PROOF (SKETCH): The restriction on unions ensures that at all times while the sequence is being processed, each set either contains at least one special vertex or is a singleton set containing an ordinary vertex.

A standard DSU data structure, U, operates on the special vertices. The ordinary vertices are kept separately. Each ordinary vertex contains a pointer, initially null, that can point to a special vertex.

Implementing the operations with this data structure is simple: unless an ordinary vertex x forms a singleton set, its pointer points to a special vertex y such that *find*(x) = *find*(y). Each operation involves $O(1)$ steps plus, possibly, an operation on U, which contains l vertices. Let k be the number of operations done on U. Then the total running time is $O(k\alpha(k, l) + m + n)$, which is $O(m\alpha(m, l) + n)$ for $k = O(m)$. □

We can implement the above algorithm within the framework of a single DSU data structure. We weight special vertices one and ordinary vertices zero. Recall that the size of a vertex is the sum of the weights of its descendents, including itself. The union-by-size rule ensures that whenever a singleton ordinary set is united with a set containing special elements, the ordinary vertex is made a child of the other root. Path compressions never change leaves into non-leaves, so each ordinary vertex is at all times either a singleton root or a child of a special vertex, as in the proof of Theorem 3.1. Furthermore, since the ordinary vertices have weight zero, they do not affect the size decisions made when uniting sets. A find on an ordinary vertex u is equivalent to a find on its parent, which is a special vertex, just as in the proof of Theorem 3.1.

3.1 Bottom-Up linking

Let a sequence of unions on U be described by a rooted, undirected *union tree*, T, each vertex of which corresponds to an element of U. The edges in T are labeled zero or one; initially, they are all labeled zero. Vertices connected by a path in T of edges labeled one are in the same set. Labeling an edge $\{v, p(v)\}$ one corresponds to uniting the sets containing v and $p(v)$. The union sequence has the *bottom-up linking property* if no edge $\{v, p(v)\}$ is labeled one until all edges in the subtree rooted at v are labeled one.

Corollary 3.2 *Let T be a union tree with l leaves and the bottom-up linking property. Then $n-1$ unions and m finds can be performed in $O(m\alpha(m,l)+n)$ time.*

PROOF (SKETCH): Let the leaves of T be classed as special and all internal vertices classed as ordinary. Since unions occur bottom-up, each non-singleton set always contains at least one leaf. □

Alstrup et al. [3] prove a variant of Corollary 3.2, with the $m\alpha(m,l)$ term replaced by $(l\log l+m)$, which suffices for their purposes. They derive the weaker result by processing long paths of unary vertices in T outside the standard set union data structure. We apply the standard set union data structure directly to T; we need only weight the leaves of T one and the internal vertices of T zero.

3.2 Bottom-Level Microtrees

Recall the definitions from Section 2 for the LCA problem. By Corollary 3.2, if we restrict the tree T to have few leaves, then the AHU LCA algorithm will take only linear time. To do this, we remove from T small subtrees, which we call microtrees, such that the remaining tree T' has a small number of leaves. We then compute the least common ancestors of pairs that are contained in a single microtree, exploiting the limited number of microtrees and sets of query pairs within the microtree. We compute the least common ancestors of all the other pairs using the AHU algorithm modified so that the DSU data structure operates only on vertices of T'. We will show that each part takes linear time.

We partition T by removing small subtrees that together contain all the leaves of T, i.e., subtrees from the bottom of T. Let T_v be the subtree of T rooted at v; let $|T_v|$ be the number of vertices in T_v. Let g be some parameter to be fixed later. We define T_v to be a *microtree* if $|T_v| \le g$ and $|T_{p(v)}| > g$; for a vertex $x \in T_v$, *micro*$(x) = T_v$ is x's microtree, and *root*(*micro*$(x)) = v$ is the *root* of its microtree. Let T' be what remains of T after removing all the microtrees. It is straightforward to compute this partition using DFS.

Each leaf in T' has more than g descendents in T, and the descendents in T of two leaves in T' form disjoint sets, so T' has $O(n/g)$ leaves.

Section 4 presents a general technique that we can apply to compute the LCA for each pair $\{x, y\} \in P$ such that *micro*(x) and *micro*(y) are defined and equal, i.e., for each pair contained in the same microtree. To compute the LCAs for all the other pairs we use the AHU algorithm, modified as follows. Initially each vertex not in any microtree is a singleton set. When we scan a vertex v, for each pair $\{v, w\}$ such that w is not in the same microtree as v and w has already been scanned, we set *lca*(v, w) to be (1) *find*(w), if w does not belong to any microtree or (2) *find*$(p($*root*$($*micro*$(w))))$, if w is in a microtree. When we backtrack from a vertex v not in any microtree we perform $union(v, p(v), p(v))$. The correctness of the algorithm follows from the following lemma and basic DFS properties.

Lemma 3.3 *For every pair $\{v, w\}$ such that v and w are in the same microtree S, lca(v, w) in T is the same as lca(v, w) in S. For every other pair $\{v, w\}$, lca(v, w) is the same as lca$(r(v), r(w))$, where $r(v) = p($root$($micro$(v)))$ or, if v is not in any microtree, $r(v) = v$.*

Since unions and finds are restricted to T' and unions occur in bottom-up order, Corollary 3.2 implies that the unions and finds take $O(p\alpha(p, n/g) + n)$ time. We fix g in the next section to make this linear.

Gabow and Tarjan [14] pioneered the use of microtrees to produce a linear-time DSU algorithm for the special case when the unions are known in advance. They partition an entire tree into microtrees. Dixon and Tarjan [10] introduce the idea of processing microtrees only at the bottom of a tree. Alstrup et al. [3] use bottom-level microtrees to speed the computation of dominators, which we address in Section 6. Corollary 3.2 offers a new analysis that demonstrates the general utility of processing microtrees only at the bottom of a tree.

4 Linear-Time, Pointer-Machine Processing of Small Graphs

Let $\mathcal{P}$ be some computation on a graph G that, based only on the structure of G, produces an output whose components are identified with vertices and edges (or arcs) in G. Let $\mathcal{T}(|G|)$ be the time to compute $\mathcal{P}$ on G on a pointer machine. Let $\mathcal{G}$ be a collection of instances of $\mathcal{P}$, each of size no greater than g. Let $N = \sum_{G\in\mathcal{G}} |G|$ be the total size of the instances. We want to apply $\mathcal{P}$ to each $G \in \mathcal{G}$.

When an instance can be encoded in one computer word, one can exploit that the number of possible distinct instances is small compared to N. By building a table, indexed by instance encoding, that contains the results of $\mathcal{P}$ for each distinct instance, each input instance $G \in \mathcal{G}$ is solved by a constant-time lookup into the table. Since $g \ll N$, the table can be constructed in $O(N)$ time and space, even when $\mathcal{T}(g) \gg g$. This approach, introduced by Gabow and Tarjan [14], as well as a variant that uses memoization, requires a RAM.

Rather than build and index a table, we identify duplicate instances in $\mathcal{G}$, compute $\mathcal{P}$ once for each distinct $G \in \mathcal{G}$, and copy the answers to the duplicates. To do so, we introduce a pointer-based radix sort to organize instance encodings.

4.1 Building Adjacency List Encodings

Let $G \in \mathcal{G}$ be some instance; each vertex and edge in G may contain $O(1)$ bits of extra information. We perform a DFS of G, building for each vertex $v \in G$ a linked list of neighbors (or successors, in the directed case). Since we traverse G in DFS order, we can build and maintain the neighbor lists via a linked list, associating a record pointing to a neighbor list with each vertex. We build an encoding for G by catenating the neighbor lists for all the $v \in G$, in order by v, using the non-DFS number 0 as a delimiter. We append the extra information in a vertex/edge to the occurrence of the respective graph component in the encoding.

Since each encoding contains no more than $2g(g+1)$ numbers, each in the range $[0, g]$, there are about g^{g^2} possible encodings. Each instance $G \in \mathcal{G}$ is described by one such encoding, implemented as a linked list requiring $O(|G|)$ space. The computation of all the encodings requires $O(N)$ time.

4.2 Identification of Duplicates via Pointer-Based Radix Sort

To locate duplicate encodings, we sort the list of encodings lexicographically. Using a variant of radix sort that sorts strings of unequal length [1], a collection of keys, each a vector of digits in the range $[0, g]$, can be sorted in time $O(Lg + \sum_i \ell_i)$, where ℓ_i is the length of the ith key and $L = \max_i\{\ell_i\}$. Since $L = O(g^2)$, sorting the encodings takes $O(g^3 + N)$ time.

Recall that standard radix sort proceeds in a series of passes: the ith pass places a key into a bucket determined by its ith rightmost digit. Short keys are implicitly left padded with 0's. Thus, when we reach the end of a key, we can consider it sorted and append it to a DONE list. Normally, a key is placed into a bucket using the ith digit to index a table, in which element j points to a linked list of keys in bucket j.

All that remains is to implement the buckets using only pointers. There are $g+1$ buckets, one for each vertex number and the delimiter 0, which we arrange in a linked list $\mathcal{B}$. We identify a vertex with its associated bucket as follows. During the DFS of each $G \in \mathcal{G}$, we maintain a pointer into $\mathcal{B}$; as we increment the current DFS number, we simultaneously move the pointer into $\mathcal{B}$ to the next bucket. We can thus associate a pointer to bucket i with the vertex numbered i during the DFS. Each datum of extra information contains $O(1)$ bits, so the corresponding bucket can be found by following $O(1)$ pointers from the head of $\mathcal{B}$. Now, when constructing the encodings, we use the bucket pointers instead of numbers as the encoding components. During the sort, we access the bucket corresponding to each vertex by following these pointers.

4.3 Computation of $\mathcal{P}$

We now process the sorted list of encodings. For each distinct encoding, we construct a corresponding canonical instance G' and compute $\mathcal{P}$ on G'. For each occurrence of the encoding, let $G \in \mathcal{G}$ be the corresponding instance. Since G and G' are isomorphic, we can traverse them in tandem, adding pointers between corresponding vertices and edges. We then assign the results of $\mathcal{P}$ to G using this mapping. Note that this approach assumes that $\mathcal{P}$ is an off-line computation.

There can be about g^{g^2} different encodings, so the total time to sort the encodings and compute $\mathcal{P}$ for each canonical instance is $O(N + g^3 + g^{g^2}\mathcal{T}(g))$. Choosing g appropriately therefore makes the entire computation take $O(N)$ time on a pointer machine. In particular, for any $\mathcal{P}$ such that $\mathcal{T}(g)$ itself is $O(g^{g^2})$, choosing $g = O(\log^{1/3} N)$ suffices.

4.4 Application to LCAs

Theorem 4.1 *The off-line LCAs algorithm of Section 3.2 runs in $O(n+p)$ time on a pointer machine.*

PROOF: We can apply the above technique to compute LCAs for the pairs contained in microtrees in linear time on a pointer machine when $g = O(\log^{1/3} n)$. Each instance is a microtree plus the queries within the microtree; we realize the queries as graph edges, marked by a bit. Setting $g = \log^{1/3} n$ results in T' having $O(n/\log^{1/3} n)$ leaves. Since unions occur in bottom-up order, Corollary 3.2 implies that the DSU operations, which are easily implemented on a pointer machine, take $O(p\alpha(p, n/\log^{1/3} n)+n) = O(p+n)$ time. □

The ability to recover the answers from the computation of $\mathcal{P}$ on the instances in $\mathcal{G}$, as we describe in Section 4.3, is subtle yet critical. Alstrup, Secher, and Spork [4] show how to compute connectivity queries on a tree T undergoing edge deletions, in linear time. They partition T into bottom-level microtrees and compute, for each vertex v in a microtree, a bit-string that encodes the vertices on the path from v to the root of its microtree. They show how to answer connectivity queries using a constant number of bitwise operations on these bit-strings and applying the Even and Shiloach decremental connectivity algorithm [11] to the upper part of T.

The Alstrup, Secher, and Spork algorithm [4] runs on a pointer machine: since the connectivity queries return yes/no answers, they need not index tables to recover the answers. In contrast, while their method can be extended to solve the off-line LCAs problem in linear time on a RAM, and even to simplify the Gabow-Tarjan linear-time DSU result [14], both of these extensions require indexing tables to map the results of the bitwise operations back to vertices in T.

In the next two sections, we apply the techniques of Sections 3 and 4 to the verification and construction of minimum spanning trees and the computation of dominators.

5 Minimum Spanning Trees

5.1 Verification

Let $G = (V, E)$ be a weighted graph, such that $c(x, y)$ is the weight of edge $\{x, y\}$. Let $T = (V, E_T)$ be a spanning tree of G. The *minimum spanning tree (MST) verification* problem is to determine whether or not T is an MST of G. For each non-tree edge $\{x, y\} \in E \setminus E_T$, there is a unique path $P(x, y)$ between x and y in T. It is well known that T is an MST of G if and only if $c(x, y) \geq c(u, v)$ for all $\{u, v\} \in P(x, y)$.

For any tree T, let $p_T(v)$ be the parent of v in T; we will drop the subscript when the meaning is clear from context. Tarjan [27] shows how to determine if T is an MST of G in $O(m\alpha(m, n) + n)$ time, by a procedure similar to the AHU LCA algorithm. He introduces a *link-eval* data structure, which, given any tree T on real-valued vertices, maintains a forest F contained in the tree, subject to the following operations. (Initially F contains no arcs.)

link(u)**:** Add arc $(p_T(u), u)$ to F.

eval(u)**:** Let r be the root of the tree containing u in F. If $u = r$, return r. Otherwise, return any vertex $x \neq r$ of minimum (or maximum) value on the path from r to u.

Using standard DSU, $n - 1$ links and m evals on an n-vertex tree T take $O(m\alpha(m, n) + n)$ time [27]. Applied to MST verification, the value of a vertex v is $c(v, p_T(v))$. Each non-tree edge $\{x, y\}$ is placed into a bucket associated with *lca*(x, y). Then a DFS of T is performed. When backtracking through vertex v, for each $\{x, y\} \in$ *bucket*(v), *eval*(x) (rsp., *eval*(y)) returns the weight of the maximum weight edge on $P(x, v)$ (rsp., $P(y, v)$). It thus suffices to compare $c(x, y)$ to *eval*(x) and *eval*(y). Finally, *link*(v) is performed.

As with LCAs, note that the links, and hence the unions, occur in bottom-up order. We cannot apply our microtree technique directly to reduce the running time of Tarjan's algorithm, however, because the edge weights require more than $O(1)$ bits each and hence preclude an efficient encoding as required by Section 4.1.

Dixon, Rauch, and Tarjan (DRT) [9] and King [18] provide linear-time MST verification algorithms on a RAM. Both require an LCA computation. DRT replaces each non-tree edge $\{x, y\}$ such that x and y are not related with edges $\{x, \textit{lca}(x, y)\}$ and $\{y, \textit{lca}(x, y)\}$, both of cost $c(x, y)$. For each non-tree edge $\{u, v\}$, therefore, we may assume that u is a proper ancestor of v.

DRT then partitions all of T into microtrees of size no more than g and replaces non-tree edges with edges whose endpoints are either in the same microtree or else connect roots of different microtrees. They verify T in linear time, using Tarjan's verification algorithm on the subtree induced by the microtree roots and a result from Komlós [19], which we describe below, to process the microtrees.

We can use bottom-level microtrees to simplify the DRT algorithm. Any replacement edge $\{u, v\}$ such that, (1) u and v are in the same microtree or, (2) both u and v are not in microtrees, remains unchanged. Otherwise, letting $r_v = \textit{root}(\textit{micro}(v))$, we replace $\{u, v\}$ with $\{r_v, v\}$ and $\{u, p(r_v)\}$, each of cost $c(u, v)$. We check that $c(p(r_v), r_v) \leq c(u, v)$; otherwise T is not an MST. T is then an MST of G if and only if T' and all the microtrees are MSTs of the corresponding induced subgraphs [9].

Duplicate replacement edges of the form $\{u, p(r_v)\}$ add $O(m)$ edges to T'. Since Tarjan's MST verification algorithm orders links bottom up, Corollary 3.2 shows that it takes $O(n + m)$ time to verify T', if $g = \log^{1/3} n$.

To verify the microtrees, we use a result of Komlós [19], as do DRT and King. Komlós' result states that there is a decision tree D, the vertices of which determine comparisons of edge costs, that will determine if a given tree T^* is an MST of a graph G^* having at most g vertices and $e \leq g^2/2$ edges. Furthermore, D has depth $O(e)$ and $2^{O(e)}$ edges and can be constructed in $g^3 2^{O(g^2)}$ time on a pointer machine. We eliminate duplicate replacement edges of the form $\{r_v, v\}$ by scanning r_v's neighbor list, keeping the minimum cost edges, so each microtree's induced graph has at most $g^2/2$ edges. We can thus apply the technique of Section 4 to compute the decision trees for all the microtrees in $O(n+m)$ time on a pointer machine, if $g = O(\log^{1/3} n)$. To verify each microtree, we need only traverse its decision tree, which takes linear time.

Theorem 5.1 *We can determine if T is an MST of G in $O(n + m)$ time on a pointer machine.*

5.2 Construction

The only part of the randomized linear expected time MST algorithm of Karger, Klein, and Tarjan [17] that up to now could not be implemented on a pointer machine was the verification of MSTs. Theorem 5.1 thus allows randomized MST construction to be performed in linear expected time on a pointer machine. The issue of a deterministic, comparison-based, linear-time MST algorithm remains open.

6 Dominators

A *flowgraph* is a directed graph $G = (V, A, r)$ with a distinguished start vertex $r = \textit{root}(G) \in V$, such that there is a path from r to each vertex in V. Vertex w *dominates* vertex v if every path from r to v includes w; w is the *immediate* *dominator of* v, denoted $w = idom(v)$, if (1) w dominates v, and (2) every other vertex x that dominates v also dominates w. Every vertex in a flowgraph has a unique immediate dominator [2,22].

Finding immediate dominators in a flowgraph is an elegant problem in graph theory, with fundamental applications in global flow analysis and program optimization [2,8,12,22]. Lengauer and Tarjan's [20] $O(m\alpha(m,n))$-time algorithm capped a long line of successive improvements [2,22,24,26] ($n = |V|$, and $m = |A|$). Harel [15] claimed a linear-time dominators algorithm, but careful examination of his abstract reveals problems with his arguments. Alstrup et al. [3] detail some of those problems and resolve them using powerful bit-manipulation based data structures.

We describe the Lengauer-Tarjan algorithm below and apply our techniques to devise a new linear-time dominators algorithm. Ours is simpler than the algorithm of Alstrup et al. [3] and is the first that can be implemented on a pointer machine. We have implemented a RAM version of our algorithm. Experimental results show that the constant factors are low; we report these results in a separate paper [5].

6.1 Lengauer-Tarjan (LT)

Let D be a DFS tree of G, rooted at r. We sometimes refer to a vertex x by its DFS number; in particular, $x < y$ means that x's DFS number is less than y's. Let $w \xrightarrow{*} v$ (rsp., $w \xrightarrow{+} v$) denote that w is an ancestor (rsp., proper ancestor) of v in D; each can also denote the actual tree path.

Let $P = (u = x_0, x_1, \ldots, x_{k-1}, x_k = v)$ be a path in G. Lengauer and Tarjan [20] define P to be a *semi-dominator path* (abbreviated *sdom path*) if $x_i > v,\ 1 \le i \le k-1$. An sdom path from u to v thus avoids all tree vertices between u and v. The *semi-dominator of vertex* v is $semi(v) = \min\{u \mid$ there is an sdom path from u to $v\}$.

Lengauer and Tarjan [20] traverse D in reverse DFS order to compute $semi(v)$ for all $v \in V$. From the semi-dominators, they then compute the immediate dominator for each vertex. They first prove that the following procedure computes semi-dominators [20, Thm. 4]. The link-eval data structure uses $semi(v)$ as the value of vertex v; initially $semi(v) \leftarrow v$.

> **For** $v \in V$ in reverse DFS order **do**
>   **For** $(w, v) \in A$ **do**
> $u \leftarrow eval(w)$
> $semi(v) \leftarrow \min\{semi(u), semi(v)\}$
>   **done**
> $link(v)$
> **done**

Additional steps then resolve the immediate dominators, based on the semi-dominators. Note that links occur bottom-up. We cannot, however, apply our techniques directly to linearize Lengauer-Tarjan, because, similarly to the edge weights in MST verification, the semi-dominator values that can possibly be assigned to vertices in a microtree are in the range $[1, n]$, precluding an efficient representation of a microtree as required by Section 4.1.

Harel [15] presents a method to restrict the range of semi-dominator values for vertices in a microtree. He then applies Gabow-Tarjan table lookup techniques [14] to process links and evals on microtrees. In short, he applies the standard Lengauer-Tarjan algorithm to the whole graph, speeding links and evals with microtrees. Alstrup et al. [3] correct problems in Harel's abstract, using Fredman and Willard's Q-heaps [13], which require a RAM, to manage the microtrees. They too use bottom-level microtrees, but they treat long paths of unary vertices specially to derive a weaker version of Corollary 3.2.

Our approach, on the other hand, is to determine for each vertex whether its idom is in its microtree and if so, the idom value. We then use the standard Lengauer-Tarjan algorithm on the rest of D, which we restrict by bottom-level partitioning to have few leaves, thereby speeding the algorithm by Corollary 3.2.

To summarize, Harel [15] and Alstrup et al. [3] take a purely data-structures approach, leaving the Lengauer-Tarjan algorithm unchanged but employing sophisticated new data structures to improve its running time. We modify the Lengauer-Tarjan algorithm so that, although it becomes slightly more complicated, simple and standard data structures suffice to implement it.

6.2 Linear-Time Dominators

As in Section 3.2, our algorithm for dominators removes small microtrees from the bottom of D. We process vertices in the microtrees using the technique of Section 4. We then apply the Lengauer-Tarjan paradigm to the rest of D. Corollary 3.2 shows that the links and evals take only linear time. The full details are available separately [5]; we summarize them below.

We extend the definitions for microtrees given in Section 3.2 so that every vertex is in a microtree. If $|T_v| \le g$ and $|T_{p(v)}| > g$, then T_v is a *non-trivial microtree*. If $|T_v| > g$, then $\{v\}$ itself forms a singleton, *trivial microtree*. (No precomputation as described in Section 4 is performed on trivial microtrees.) Otherwise, v is a non-root vertex in a non-trivial microtree. The definitions of $micro(v)$ and $root(micro(v))$ extend easily.

We define $P = (u = x_0, x_1, \ldots, x_{k-1}, x_k = v)$ to be an *external dominator path* (abbreviated *xdom path*) if P is an sdom path and $x_0, \ldots, x_{k-1} \notin micro(v)$. An external dominator path is simply a semi-dominator path that resides wholly outside the microtree of the target vertex (until it hits the target vertex). The *external dominator of vertex* v is $xdom(v) = \min\{\{v\} \cup \{u \mid$ there is an xdom path from u to $v\}\}$. Note that for any vertex v that forms a singleton microtree, $xdom(v) = semi(v)$.

We define $P = (u = x_0, x_1, \ldots, x_{k-1}, x_k = v)$ to be a *pushed external dominator path* (abbreviated *pxdom path*) if

$x_i \geq root(micro(v))$, $1 \leq i \leq k-1$. Since non-trivial microtrees occur only at the bottom of D, a pxdom path cannot exit and re-enter $micro(v)$. To do so would require following a back arc to a proper ancestor of $root(micro(v))$. A pxdom path to v is thus (a) an xdom path outside $micro(v)$ catenated with (b) a path inside $micro(v)$. Either (a) or (b) may be null. The *pushed external dominator of vertex* v is $pxdom(v) = \min\{u \mid$ there is a pxdom path from u to $v\}$. Since the arc $(p_D(root(micro(v))), root(micro(v)))$ catenated with the tree path $root(micro(v)) \xrightarrow{*} v$ forms a pxdom path to v, $pxdom(v) \notin micro(v)$.

Lemma 6.1 *For any v that forms a singleton microtree, $pxdom(v) = semi(v)$.*

PROOF (SKETCH): Let P be a pxdom path to v. Since $pxdom(v)$ is the minimum start node among all such paths, we can assume without loss of generality that v occurs only as the last vertex in P. Therefore, if $micro(v) = \{v\}$, P is a semi-dominator path. □

We also use the following facts from Lengauer and Tarjan [20] to prove the correctness of our algorithm.

Lemma 6.2 (LT Lemma 1) *If $v \leq w$ then any path from v to w in G must contain a common ancestor of v and w in D.*

Lemma 6.3 (LT Lemma 4) *For any $v \neq r$, $idom(v) \xrightarrow{*} semi(v)$.*

Lemma 6.4 (LT Lemma 5) *Let w, v satisfy $w \xrightarrow{*} v$. Then $w \xrightarrow{*} idom(v)$ or $idom(v) \xrightarrow{*} idom(w)$.*

Lemma 6.5 (LT Theorem 2) *Let $w \neq r$. Suppose every u for which $semi(w) \xrightarrow{+} u \xrightarrow{*} w$ satisfies $semi(u) \geq semi(w)$. Then $idom(w) = semi(w)$.*

Lemma 6.6 (LT Theorem 3) *Let $w \neq r$, and let u be a vertex for which $semi(u)$ is minimum among vertices u satisfying $semi(w) \xrightarrow{+} u \xrightarrow{*} w$. Then $semi(u) \leq semi(w)$ and $idom(u) = idom(w)$.*

The following lemma is analogous to Lemma 6.3.

Lemma 6.7 *$idom(v) \notin micro(v) \Rightarrow idom(v) \xrightarrow{*} pxdom(v)$.*

PROOF: Let $u = pxdom(v)$. As observed above, $u \notin micro(v)$. By definition of pxdom, there is a path from u to v that avoids all vertices (other than u) on the tree path $u \xrightarrow{*} p_D(root(micro(v)))$. Therefore, if $idom(v) \notin micro(v)$, $idom(v)$ cannot lie on that tree path. □

Our dominators algorithm proceeds roughly as follows. We compute a DFS tree D of G and partition D into microtrees. Using the technique of Section 4, we determine for each v if $idom(v) \in micro(v)$ and, if so, determine the actual value $idom(v)$. We then compute $pxdom(v)$ for all v. In reverse DFS order, based on $pxdom(v)$ and whether or not $idom(v) \in micro(v)$, we compute $idom(v)$ directly or determine an ancestor u of v such that $idom(v) = idom(u)$. The computation and use of pxdoms essentially applies the Lengauer-Tarjan algorithm to the subtree D' of D consisting of the trivial microtree roots. Since D' has $O(n/g)$ leaves, the links and evals take linear time, by Corollary 3.2.

6.3 Computing Internal Dominators

Let T be a microtree. Let $G(T)$ be the subgraph of G induced by vertices of T. Let $aug(T)$ be the graph $G(T)$ plus (1) a new vertex $t = root(aug(T))$ and (2) a *blue arc* (t, v) for each $v \in T$ such that there exists an arc (u, v) for some $u \notin T$. Vertex t represents the contraction of $G \setminus T$, ignoring all arcs that exit T.

We define the *internal immediate dominator of vertex* x, $iidom(x)$, to be the immediate dominator of x in $aug(micro(x))$. We can compute the iidoms for all vertices using the technique of Section 4. The following two lemmas show how to determine from $iidom(v)$ if $idom(v) \in micro(v)$ and, if so, the value of $idom(v)$.

Lemma 6.8 *Let $T = micro(x)$ and $t = root(aug(T))$. Then $iidom(x) \neq t \Rightarrow idom(x) = iidom(x)$.*

PROOF (SKETCH): Let $y = iidom(x)$ and $z = idom(x)$ such that $y \neq t$ and $y \neq z$. If $z < y$, then in the full graph G, there exists a path P from z to x that avoids y, and from P we can derive a path P' in $aug(T)$ from some $z' \in \{t, z\}$ to x that avoids y, contradicting the assumption that $y = iidom(x)$. If $y < z$, then there is a path P in $aug(T)$ from y to x that avoids z. By hypothesis, $y \neq t$, so P contains no blue arcs. Therefore, P is also a path in G, contradicting that $z = idom(x)$. □

Lemma 6.9 *Let $T = micro(x)$ and $t = root(aug(T))$. Then $iidom(x) = t \Rightarrow idom(x) \notin micro(x)$.*

PROOF (SKETCH): Suppose $idom(x) = z \in micro(x)$ but $iidom(x) = t$. Then there is a path P in $aug(T)$ from t to x that avoids z. From P we can demonstrate a similar path in the original graph, contradicting the claim that $z = idom(x)$. □

6.4 Computing Pushed External Dominators

To compute pxdoms, we process microtrees T in reverse DFS order, as follows. Initially, $label(v) \leftarrow v$, and $label(v)$ is the value for vertex v in the link-eval data structure. Let $EN(v) = \{x \mid (x, v) \in A, x \notin micro(v)\}$ be the vertices outside $micro(v)$ with arcs to v.

1. For $v \in T$:
   (a) $B = \{label(x) \mid x \in EN(v)\}$;
   (b) $C = \{label(eval(p_D(root(micro(x))))) \mid x \in EN(v), x \not\xrightarrow{*} v\}$;
   (c) $label(v) \leftarrow \min\{\{v\} \cup B \cup C\}$.

2. For $v \in T$, *push* to v. Let Y be the set of all vertices in T from which there is a path to v consisting only of arcs in $G(T)$. Set $\mathit{label}(v) \leftarrow \min_{y \in Y}\{\mathit{label}(y)\}$. (This can be done by computing strongly connected components (SCCs) and processing them in topological order.)

3. If T is a trivial microtree, then $\mathit{link}(\mathit{root}(T))$.

Lemma 6.10 *Upon termination of the above procedure, $\mathit{label}(v) = \mathit{pxdom}(v)$ for all v.*

PROOF (SKETCH): Consider the processing of a microtree T, and assume by induction that all vertices in microtrees that precede T in reverse DFS order are correctly labeled by their pxdoms. For any $v \in T$, we show that after Step 2, $\mathit{label}(v) \le \mathit{pxdom}(v)$ and $\mathit{pxdom}(v) \le \mathit{label}(v)$.

Let $\mathit{pxdom}(v) = w$, and consider any pxdom path P from w to v. Let (y, x) be the arc in P that crosses into T; i.e., $y \notin T$, and $x \in T$. Let $T' = \mathit{micro}(y)$. Let z be the least vertex in P on the tree path $\mathit{lca}(v, y) \xrightarrow{*} y$. The prefix of P from w to z is a semi-dominator path to z. If $z \in T'$, then from P we can derive a pxdom path from w to y, so $\mathit{label}(y) \le w$ by induction. Otherwise, $z \notin T'$; in this case, $\mathit{label}(\mathit{eval}(p_D(\mathit{root}(\mathit{micro}(y))))) \le w$ by induction. Thus, $\mathit{label}(x) \le w$ after Step 1, and $\mathit{label}(v) \le w = \mathit{pxdom}(v)$ after Step 2.

Conversely, for any $x \in T$ such that there is a path in $G(T)$ from x to v, consider any $y \in \mathit{EN}(x)$; by induction there is a pxdom path P' from $\mathit{label}(y)$ to y. We can augment P' into a pxdom path to v, using arc (y, x) and the path in $G(T)$ from x to v, so $\mathit{pxdom}(v) \le \mathit{label}(y)$. Similarly, if $y \not\xrightarrow{*} v$, let $z = \mathit{eval}(p_D(\mathit{root}(\mathit{micro}(y))))$; by induction there is a pxdom path P'' from $\mathit{label}(z)$ to z, and we can augment P'' into a pxdom path to v, using tree path $z \xrightarrow{*} y$, arc (y, x), and the path in $G(T)$ from x to v. Thus, $\mathit{pxdom}(v) \le \mathit{label}(z)$. Therefore, $\mathit{pxdom}(v) \le \mathit{label}(v)$ after Step 2. □

Pxdoms are non-increasing along paths inside a microtree. We thus perform evals only on parents of microtree roots in Algorithm IDOM (see Figure 1): the pxdom pushing in Step 2 effectively substitutes for evals on vertices inside microtrees.

6.5 Computing Dominators

To complete our algorithm, we rely on the following.

Lemma 6.11 *For any v, there exists a $w \in \mathit{micro}(v)$ such that*

1. *$w \xrightarrow{*} v$,*
2. *$\mathit{pxdom}(v) = \mathit{pxdom}(w)$,*
3. *$\mathit{pxdom}(w) = \mathit{semi}(w)$, and*
4. *$\mathit{iidom}(w) = \mathit{root}(\mathit{aug}(\mathit{micro}(w)))$.*

PROOF (SKETCH): The proof proceeds as follows. We first find an appropriate vertex w on tree path $\mathit{root}(\mathit{micro}(v)) \xrightarrow{*} v$. We show that $\mathit{semi}(w) = \mathit{pxdom}(v)$ and $\mathit{pxdom}(w) = \mathit{pxdom}(v)$. This resolves postulates (1)–(3). Finally, we prove that $\mathit{idom}(w) \notin \mathit{micro}(x)$, which implies postulate (4)

Let $x = \mathit{pxdom}(v)$, and consider the pxdom path P from x to v. Let w be the least vertex in P on the tree path $\mathit{root}(\mathit{micro}(v)) \xrightarrow{*} v$. The prefix P' of P from x to w must be a semi-dominator path. Otherwise, there is some $y < w$ on P'; by definition of $\mathit{pxdom}(v)$, $y \in \mathit{micro}(x)$. Applying Lemma 6.2 to y and w yields a z on P' such that $z \xrightarrow{+} w$ and $z \in \mathit{micro}(v)$. This contradicts the assertion that w is the least vertex in P on tree path $\mathit{root}(\mathit{micro}(v)) \xrightarrow{*} v$. Therefore $\mathit{semi}(w) \le x$. If $\mathit{semi}(w) < x$, however, then $\mathit{pxdom}(v) \le \mathit{pxdom}(w) \le \mathit{semi}(w) < x$. Thus, $\mathit{pxdom}(v) = \mathit{pxdom}(w) = x$, and $\mathit{pxdom}(w) = \mathit{semi}(w)$.

Since $\mathit{pxdom}(w) \notin \mathit{micro}(w)$, $\mathit{pxdom}(w) = \mathit{semi}(w)$ implies that $\mathit{semi}(w) \notin \mathit{micro}(w)$. Thus, by Lemma 6.3, $\mathit{idom}(w) \notin \mathit{micro}(w)$, and by Lemma 6.8, $\mathit{iidom}(w) = \mathit{root}(\mathit{aug}(\mathit{micro}(w)))$. □

Lemma 6.12 *Let w, v be vertices in a microtree T such that*

1. *$w \xrightarrow{*} v$,*
2. *$\mathit{pxdom}(w) = \mathit{pxdom}(v)$, and*
3. *$\mathit{iidom}(v) = \mathit{iidom}(w) = \mathit{root}(\mathit{aug}(T))$.*

Then $\mathit{idom}(v) = \mathit{idom}(w)$.

PROOF (SKETCH): Condition (3) and Lemma 6.9 imply that $\mathit{idom}(v), \mathit{idom}(w) \notin T$. Thus, by Lemma 6.4, $\mathit{idom}(v) \le \mathit{idom}(w)$. If $\mathit{idom}(v) < \mathit{idom}(w)$, then there is a path P from $\mathit{idom}(v)$ to v that avoids $\mathit{idom}(w)$. P contains a semi-dominator subpath P' from some $y < \mathit{idom}(w)$ to some $x > \mathit{idom}(w)$ such that $x \xrightarrow{*} v$. If $x \in \mathit{idom}(w) \xrightarrow{+} w$, then $\mathit{idom}(w) \le y$. If $x \in w \xrightarrow{*} v$, then $\mathit{pxdom}(v) \le y < \mathit{pxdom}(w)$. (By Lemma 6.7, $\mathit{idom}(w) \le \mathit{pxdom}(w)$.) So no such P' can exist. □

We can now compute idoms by Algorithm IDOM, given in Figure 1. For each $v \in D$, IDOM either computes $\mathit{idom}(v)$ or determines a proper ancestor u of v such that $\mathit{idom}(v) = \mathit{idom}(u)$. IDOM uses a second link-eval data structure, with pxdoms as vertex values. At the beginning of IDOM, no links have been done.

Theorem 6.13 *Algorithm* IDOM *correctly assigns immediate dominators.*

PROOF (SKETCH): Assigning $\mathit{idom}(v)$ to be $\mathit{iidom}(v)$ if $\mathit{iidom}(v) \in \mathit{micro}(v)$ is correct, by Lemma 6.8. Assume then that $\mathit{iidom}(v) \notin \mathit{micro}(v)$, and thus $\mathit{idom}(v) \notin \mathit{micro}(v)$ by Lemma 6.9.

Consider the processing of vertex v in bucket(u), and assume first that $\mathit{pxdom}(v) = \mathit{semi}(v) = u$. Let u' be the

Algorithm IDOM
  For $v \in D$ in reverse DFS order **do**
    Process(v)
  done.
  For $u \in D$ such that $\{u\}$ is a trivial microtree, in reverse DFS order **do**
    Process(*bucket*(u))
    link(u)
  done.

$\mathrm{Process}(v)$
  If *iidom*$(v) \in$ *micro*(v) **then**
    idom$(v) \leftarrow$ *iidom*(v)
  else
    add v to *bucket*(*pxdom*(v))
  endif.

$\mathrm{Process}$(*bucket*(u))
  For $v \in$ *bucket*(u) **do**
    If $u = p_D$(*root*(*micro*(v))) **then**
      $z \leftarrow v$
    else
      $z \leftarrow$ *eval*(p_D(*root*(*micro*(v))))
    endif.
    If *pxdom*$(z) = u$ **then**
      idom$(v) \leftarrow u$
    else
      idom$(v) =$ *idom*(z)
    endif.
  done.

Figure 1: Algorithm IDOM.

child of u on tree path $u \xrightarrow{+} v$. We claim that z is the vertex on tree path $u' \xrightarrow{*} v$ with minimum semi and that *pxdom*$(z) =$ *semi*(z). Assuming that this claim is true, if *pxdom*$(z) = u$, then by Lemma 6.5 *idom*$(v) =$ *semi*$(v) = u$, and if *pxdom*$(z) < u$, then by Lemma 6.6 *idom*$(v) =$ *idom*(z).

Observe that for any $w \in$ *micro*(v) such that $w \xrightarrow{*} v$, *semi*$(v) =$ *pxdom*$(v) \leq$ *pxdom*$(w) \leq$ *semi*(w). If $u = p_D$(*root*(*micro*(v))), then $u' =$ *root*(*micro*(v)), $z = v$, and the claim holds. On the other hand, if $u \xrightarrow{+} p_D$(*root*(*micro*(v))), then $z =$ *eval*(p_D(*root*(*micro*(v)))) is the vertex on the tree path $P = u' \xrightarrow{*} p_D$(*root*(*micro*$(v)$)) of minimum pxdom. The claim holds, since (1) *pxdom*$(u') \leq u =$ *pxdom*(v), and (2) *pxdom*$(y) =$ *semi*(y) for all $y \in P$ (by Lemma 6.1).

In the case that *pxdom*$(v) \neq$ *semi*(v), we can apply Lemmas 6.11 and 6.12 to find a $w \in$ *micro*(v) such that $w \xrightarrow{*} v$, *pxdom*$(v) =$ *pxdom*$(w) =$ *semi*(w), and *idom*$(v) =$ *idom*(w). Thus, IDOM places v and w into the same bucket, and since IDOM computes *idom*(w) correctly (as above), it also computes *idom*(v) correctly. □

6.6 Analysis

Theorem 6.14 *Algorithm* IDOM *can be implemented on a pointer machine in* $O(n+m)$ *time.*

PROOF: Computation of iidoms can be implemented on a pointer machine in linear time, for $g = O(\log^{1/3} n)$, by the technique in Section 4. Excluding the time to perform the $O(n)$ links and $O(m)$ evals, the computation of pxdoms and idoms takes $O(n+m)$ time, including time to compute SCCs.

Consider the subtree T of D induced by the trivial microtree roots. All the links and evals are performed on vertices of T. T has $O(n/g) = O(n/\log^{1/3} n)$ leaves. The links are performed bottom-up, due to the reverse-DFS processing order. By Corollary 3.2, the link-eval time is thus $O(m\alpha(m, n/\log^{1/3} n) + n) = O(n+m)$. □

7 Conclusion

We have presented two new tools for designing efficient algorithms on pointer machines: DSU with bottom-up linking, and pointer-based radix sort processing of small graphs. We have combined these tools with bottom-level microtrees to produce the first linear-time pointer-machine algorithms for off-line LCAs, MST verification, randomized MST construction, and computing dominators in a flowgraph. Our algorithms are simpler than the corresponding RAM algorithms. We have implemented a RAM version of our dominators algorithm. Experimental results, which we report separately [5], show that it has low constant factors.

Acknowledgement

We thank Bob Tarjan for helpful discussions.

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

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

[3] S. Alstrup, D. Harel, P. W. Lauridsen, and M. Thorup. Dominators in linear time. Manuscript submitted, 1997.

[4] S. Alstrup, J. P. Secher, and M. Spork. Optimal on-line decremental connectivity in trees. *IPL*, 64(4):161–4, 1997.

[5] A. L. Buchsbaum, H. Kaplan, A. Rogers, and J. R. Westbrook. A new, simpler linear-time dominators algorithm. Technical Report 97.31.2, AT&T Labs–Research, 1998. Submitted for publication.

[6] A. L. Buchsbaum, R. Sundar, and R. E. Tarjan. Data-structural bootstrapping, linear path compression, and catenable heap-ordered double-ended queues. *SIAM J. Comp.*, 24(6):1190–1206, 1995.

[7] B. Chazelle. A faster deterministic algorithm for minimum spanning trees. In *Proc. 38th IEEE FOCS*, pages 22–31, 1997.

[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 Trans. Prog. Lang. Sys.*, 13(4):451–90, 1991.

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

[10] B. Dixon and R. E. Tarjan. Optimal parallel verification of minimum spanning trees in logarithmic time. *Algorithmica*, 17:11–8, 1997.

[11] S. Even and Y. Shiloach. An on-line edge deletion problem. *J. ACM*, 28(1):1–4, 1981.

[12] J. Ferrante, K. Ottenstein, and J. Warren. The program dependency graph and its uses in optimization. *ACM Trans. Prog. Lang. Sys.*, 9(3):319–49, 1987.

[13] M. L. Fredman and D. E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *JCSS*, 48:533–51, 1994.

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

[15] D. Harel. A linear time algorithm for finding dominators in flow graphs and related problems. In *Proc. 17th ACM STOC*, pages 185–94, 1985.

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

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

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

[19] J. Komlós. Linear verification for spanning trees. *Combinatorica*, 5(1):57–65, 1985.

[20] T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Prog. Lang. Sys.*, 1(1):121–41, 1979.

[21] M. Loebl and J. Nesetril. Linearity and unprovability of set union problem strategies I. Linearity of strong postorder. *J. Alg.*, 23:207–20, 1997.

[22] E. S. Lorry and V. W. Medlock. Object code optimization. *C. ACM*, 12(1):13–22, 1969.

[23] J. M. Lucas. Postorder disjoint set union is linear. *SIAM J. Comp.*, 19(5):868–82, 1990.

[24] P. W. Purdom and E. F. Moore. Algorithm 430: Immediate predominators in a directed graph. *C. ACM*, 15(8):777–8, 1972.

[25] B. Schieber and U. Vishkin. On finding lowest common ancestors: Simplification and parallelization. *SIAM J. Comp.*, 17(6):1253–62, 1988.

[26] R. E. Tarjan. Finding dominators in directed graphs. *SIAM J. Comp.*, 3(1):62–89, 1974.

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

[28] R. E. Tarjan. A class of algorithms which require nonlinear time to maintain disjoint sets. *JCSS*, 18(2):110–27, 1979.

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