Dynamic Trees as Search Trees via Euler Tours, Applied to the Network Simplex Algorithm

Robert E. Tarjan*

Research Report CS-TR-503-95

September, 1995

Abstract

The *dynamic tree* is an abstract data type that allows the maintenance of a collection of trees subject to joining by adding edges (*linking*) and splitting by deleting edges (*cutting*), while at the same time allowing reporting of certain combinations of vertex or edge values. For many applications of dynamic trees, values must be combined along paths. For other applications, values must be combined over entire trees. For the latter situation, we show that an idea used originally in parallel graph algorithms, to represent trees by Euler tours, leads to a simple implementation with a time of $O(\log n)$ per tree operation, where n is the number of tree vertices. We apply this representation to the implementation of two versions of the network simplex algorithm, resulting in a time of $O(\log n)$ per pivot, where n is the number of vertices in the problem network.

1 Introduction

Consider a collection of unrooted tress, each initially a single vertex and no edges, on which two structural update operations are allowed:

*Department of Computer Science, Princeton University, Princeton, NJ 08544 and NEC Research Institute, 4 Independence Way, Princeton, NJ 08540. Research at Princeton University partially supported by the National Science Foundation, Grant No. CCR-8920505, and the Office of Naval Research, Contract No. N0014-91-J-1463.

link($\{v, w\}$): Combine the trees containing vertices v and w by adding the edge $\{v, w\}$. This operation does nothing if v and w are already in the same tree.

cut($\{v, w\}$): Break the tree containing edge $\{v, w\}$ in two by deleting the edge $\{v, w\}$. This operation does nothing if $\{v, w\}$ is not an existing tree edge.

Suppose further that each tree vertex v has an associated real value, denoted by *val*(v), and that the following operations on values are allowed:

find-val (v): return *val*(v).

find-min-val (v): return a vertex of minimum value in the tree containing vertex v.

change-val(v, x): set *val*(v) equal to x.

add-val(v, x): add x to *val*(w) for each vertex w in the tree containing v.

Section 2 of this paper presents a simple implementation of these six tree operations with a running time of $O(\log n)$ per operation, where n is the number of vertices in the tree or trees involved in the operation. The idea used is to linearize each tree by constructing an *Euler tour* that traverses each edge once in each direction and includes one stop at each vertex, and to represent such a tour by a search tree. Linking and cutting of trees translate into simple combinations of catenation and splitting operations on tours, and on the corresponding search trees. The $O(\log n)$ time bound per tree operation is amortized if splay trees [21] are used in the representation and worst-case if balanced search trees are used. Section 3 describes the use of the Euler tour structure in the implementation of two versions of the network simplex algorithm.

The Euler Tour representation of trees was originally used in fast parallel graph algorithms [25]. Later, Miltersen et al. [18] adapted it to represent trees subject to linking and cutting but without vertex values, as did Henzinger and King [16] independently to represent trees whose vertices have associated lists.

Additional related work deals with a class of dynamic trees in which vertex or edge values are combined along paths, rather than over an entire tree. Dynamic trees of this kind arise in various network flow algorithms [10, 11, 12, 13, 20, 22, 24] and in other settings[4, 6]. Two different representations of such trees have been proposed. The first, by Sleator and Tarjan [20, 21, 24] decomposes each tree into vertex-disjoint paths and represents these paths by search trees, either biased search trees [3] or splay trees [21]. With the former representation the time per tree operation can be made $O(\log n)$ in the worst case; with the latter representation the time per tree operation is $O(\log n)$ amortized over a worst-case sequence of operations. Frederickson [7, 8] proposed a rather different representation, called the *topology tree*, which is related to the rake and compress operations used in parallel tree processing [17]. This representation, too, has an $O(\log n)$ worst-case time bound per tree operation.

Both the Sleator-Tarjan representation and the Frederickson representation can be applied to the problem we consider here, achieving the same $O(\log n)$ time bound (see e.g. [10]). But we regard these solutions are inferior, for two reasons. First, both data structures must be extended to handle tree vertices of ordinary degree. Second, both structures, even without the unbounded degree extension, are noticably more complicated than the Euler tour structure, which ultimately is just a straightforward application of search trees, and is likely to be easier to implement and more efficient in practice.

2 Trees as Tours and Tours as Search Trees

The representation we describe in this section is a slight variant of the one proposed by Mittersen et al. [18] and independently by Henzinger and King [16] for two somewhat simpler applications. To represent a dynamic tree T, we replace each edge $\{v, w\}$ of T by two arcs (directed edges) (v, w) and (w, v), and add a loop (v, v) for each vertex v. The result is a directed graph such that each vertex has in-degree equal to its out-degree. Such a graph has at least one, and in general many, *Euler tours*: cycles that contain each arc exactly once. We represent the tree by one such tour, broken at an arbitrary place to make it into a list of the arcs.

With this representation, linking and cutting each translate into a fixed set of catenation and splitting operations on lists. Specifically, suppose we wish to perform *link* $(\{v, w\})$. Let T_1 and T_2 be the trees containing v and w respectively, and let L_1 and L_2 be the lists representing T_1 and T_2. We split L_1 just after (v, v), into lists L_1^1, L_1^2, and we split L_2 just after (w, w), into L_2^1, L_2^2. Then we form the list representing the combined tree by catenating the six lists L_1^2, $L_1^1, [(v, w)]$, L_2^2, L_2^1, $[(w, v)]$ in order. Thus linking takes two splits and five catenations; two of the latter are the special case of catenation with singleton lists.

Similarly, suppose we wish to perform *cut*$(\{v, w\})$. Let T be the tree containing $\{v, w\}$, represented by list L. We split L before and after (v, w) and (w, v), into L^1, $[(v, w)]$, L^2, $[(w, v)]$, L^3 (or symmetrically L^1, $[(w, v)]$, L^2, $[(v, w)]$, L^3. The lists representing the two trees formed by the cut are L^2 and the list formed by catenating L^1 and L^3. Thus cutting takes four splits (of which two are the special case of splitting off one element) and one catenation.

By using a simple doubly-linked list representation, we can implement *link* and *cut* to take constant time; indeed, if we make the lists circular, we can save a few pointer updates. But this begs the question of how to handle vertex values. With the representation just presented, both *find-min-val* and *add-val* take time proportional to the tree size, since each requires a scan of the entire corresponding list.

We avoid this inefficiency by representing each list as a search tree. For definiteness, we shall describe a solution based on splay trees [21], a form of self-adjusting binary search tree, although any kind of search tree, such as red-black trees [15, 24], AVL trees [1], or B-trees [2], will suffice. Since the required operations on search trees are well-known and straightforward, we shall be very sketchy in the presentation. Detailed discussions of splay trees can be found in [21, 24].

Each arc in an Euler tour list becomes a node in the splay tree representing the list. Linear order in the list corresponds to symmetric order in the tree. Each arc has an associated value. A loop (v, v) has value $val(v)$; a nonloop (v, w) has value infinity. To handle *add-val* operations efficiently, we store values implicitly, in difference form. Specifically, a node x of a splay tree has stored with it

$$dif\text{-}val(x) = \begin{cases} val\ (x) \text{ if } x \text{ is a splay tree root} \\ val\ (x) - val\ (p(x)) \text{ if } x \text{ is a nonroot, where } p(x) \text{ is the parent of } x \end{cases}$$

Implementation of the various dynamic tree operations relies on *splaying*, which is the fundamental restructuring operation on splay trees. Splaying moves a designated node to the root of the splay tree by performing a sequence of local restructurings called *rotations*. The amortized time for a splay operation on an n-node tree is $O(\log n)$, including the time to update *dif-val* and *dif-min-val*. Each of the four dynamic tree operations *find-val*, *find-min-val*, *change-val*,*add-val* takes one splay plus a constant amount of additional work; in the case of *find-min-val* the path along which the splaying takes place is found by a search that is guided by following zero values of *dif-min-val* (see [21]). A list catenation or splitting operation also requires a single splay plus a constant amount of additional work. The implementation details of these operations can be found in [21, 24]. Thus all six dynamic tree operations can be performed in $O(\log n)$ amortized time.

Thus in two steps we have obtained a representation of dynamic trees as search trees with an amortized time bound of $O(\log n)$ per dynamic tree operation. If we use balanced search trees in place of splay trees, the $O(\log n)$ time bound becomes worst-case instead of amortized.

3 Implementation of Network Simplex Algorithms

We use the Euler tour data structure to implement two versions of the primal network simplex algorithm. The first use is in the Goldfarb-Hao algorithm for the maximum flow problem [14]. The second is on Orlin's new strongly polynomial algorithm for the minimum-cost flow problem [19]. In each case, we obtain an amortized time bound of $O(\log n)$ per pivot. Both algorithms require two uses of dynamic trees. One is to maintain the residual capacities of arcs. For this purpose a variant of one of the Sleator-Tarjan dynamic tree implementions [20, 21] that allows two values per arc suffices. The required changes to the Sleator-Tarjan structure are specified in detail in [22], as is the use of this structure in the network simplex algorithm applied to the minimum-cost flow problem. The use of this structure in the Goldfarb-Hao algorithm is described by Goldberg, et al. [10].

The second need for dynamic trees is to maintain what are in effect dual variables; specifically, shortest path lengths in the case of the Goldfarb-Hao algorithm and "potentials" in the case of the Orlin algorithm. We shall describe how to use the Euler tour structure (in place of a modified Sleator-Tarjan structure) to maintain those values. We assume familiarity with the relevant parts of [10, 14, 19]; the reader seeking a complete understanding of the network simplex algorithm in these two settings should consult these works.

We base our discussion of the Goldfarb-Hao algorithm on the presentation in [10]. The algorithm maintains a pair of trees S, Z, with a designated sink vertex t in Z. Each vertex has an associated label, which is a shortest path distance. Each pivot (the elementary step of the algorithm) requires performing the following operations on S, Z, and the vertex labels:

1. Find a vertex w of minimum label in tree Z.

2. Link S and Z into a tree T by adding an edge $\{v, w\}$.

3. Cut T into a new pair S, Z by deleting some edge $\{x, y\}$.

4. Update the labels of certain vertices.

We use the data structure of Section 2, with the value of each vertex being its label. Step 1 takes a *find-min-val* operation, step 2 a *link*, step 3 a *cut*, and step 4 one *change-val* per updated label. The total number of label updates over the entire algorithm is $O(nm)$, as is the number of pivots [10, 14], where n and m are the numbers of vertices and edges in the problem network, respectively. It follows that the amortized time per pivot is $O(\log n)$, and the total running time of the algorithm is $O(nm \log n)$. This matches the bound of Goldberg, et al. for the Goldfarb-Hao algorithm but considerably simplifies one part of the implementation. In this application the *add-val* operation is unnecessary, which allows us to simplify the representation of Section 2 if we wish by storing vertex values explicitly rather than in difference form.

The use of the Euler tour structure in Orlin's algorithm [19] is more complicated. In particular, we must maintain four separate values for each vertex, instead of just one, and we need one additional data structure, a heap for each vertex. Orlin's algorithm maintains a *basis tree* T that spans the vertices of the problem network. Tree T has a designated root vertex v; all edges of T are regarded as being directed toward the root. The algorithm maintains a real-valued *potential* π_i for each vertex i. Each arc (i, j) in the network, including the arcs of T and their reversals, has a *cost* c_{ij} that is part of the problem specification and a *reduced cost* $c_{ij}^{\pi} = c_{ij} - \pi_i + \pi_j$. (This definition of reduced cost uses the negative of the potential as compared to some other such definitions; see e.g. [12, 13, 24].) The algorithm maintains the property that for each tree arc (i, j), $c_{ij}^{\pi} \leq 0$.

The algorithm consists of a sequence of *phases*. It maintains a *scale factor* ϵ that remains constant during a phase and decreases between phases. We shall discuss the performance of a single phase of the algorithm. A phase begins with a set N^* initialized to contain all the vertices and ends when N^* is empty.

Let us call an arc (i, j) of T a *zero arc* if $c_{ij}^{\pi} = 0$. Our implementation maintains a collection of trees, called *zero trees*, the edges of which correspond to some of the zero arcs of T. (Not all zero arcs of T need be represented in the zero trees.) We shall denote by Z the set of arcs of T represented in the zero trees. In an abuse of notation, we shall occasionally use T to denote the set of arcs in the basis tree. Then $T - Z$ is the set of basis arcs not represented in the zero trees.

Each vertex has four associated values. (Thus we need four sets of the operations *find-val*, *find-min-val*, *change-val*, and *add-val*, although only a subset of the sixteen possible operations are actually used.) The first value for a vertex i is a bit called its *flag* that is zero if π_i has not changed during the phase and not all edges (i, k) have been examined for admissibility, and one otherwise. (We discuss admissible arcs below.) The second value for i is just π_i itself. The third value for i, called its *offset*, is in general the negative of the difference between π_i and the next larger integer multiple of $\epsilon/4$. If π_i is exactly a multiple of $\epsilon/4$, the offset is either $\epsilon/4$ or 0, depending on whether for the current value of π_i the algorithm has finished examining i for outgoing admissible arcs. (See below.) The fourth value for i, called its *mu-value*, is $\min\{c^{\pi}_{ik} | (i, j) \in T - Z\}$. The minimum element in the heap is the mu-value for i. Values in each heap are stored in difference form. This allows adding an increment x to all elements in a heap in constant time. We can use any standard heap implementation that supports the operations of insertion, deletion, and finding the minimum in time logarithmic in the heap size; see e.g. [24]. The article [9] includes a discussion of storing heap values in difference form.

By representing the zero trees using the Euler tour structure and maintaining the auxiliary heaps discussed above, it is straightforward to implement the part of Orlin's algorithm that deals with potentials. At the beginning of a phase, all vertices have a flag of zero. A single step within a phase proceeds as follows.

1. Let S be the zero tree containing the root vertex v. Find a vertex i in S with zero flag; if there is no such vertex go to 2. Find an admissible arc (i, k) and pivot on it, ending the step; if there is no such arc, delete i from N^*, set the flag of i equal to one, and set the offset of i equal to $\epsilon/4 - \pi_i(\bmod\ \epsilon/4)$, ending the step.

2. (No vertex in S has a zero flag.) Find the minimum offset Δ_1 and the minimum mu-value Δ_2 of the vertices in S; find a vertex i with offset Δ_1 and a vertex j with mu-value Δ_2. If $\Delta_1 \leq \Delta_2$, go to 3; otherwise, go to 4.

3. ($\Delta_1 \leq \Delta_2$) Add Δ_1 to the potentials of all vertices in S and subtract Δ_1 from the offsets of all vertices in S. Find an admissible arc (i, k) and pivot on it, ending the step. If there is no such arc, reset the offset of i to $\epsilon/4$, ending the step.

4. ($\Delta_2 < \Delta_1$) Add Δ_2 to the potentials of all vertices in S and subtract Δ_2 from the offset of all

vertices in S. Find an arc $(l, j) \in T - S$ with $c^{\pi}_{lj} = 0$. (Such an arc must exist after updating the potentials.) Link the zero trees containing l and j by adding edge $\{l, j\}$, ending the step.

The algorithm repeats steps until N^* becomes empty, at which time the phase ends. We make a few additonal comments about the implementation. The search for admissible arcs in 1 and 3 uses the *current arc* mechanism of Goldberg and Tarjan [11], as discussed by Orlin [19]. A *pivot* on (i, k) consists of adding the arc (i, k) to the basis tree T, deleting some arc (j, l) on the resulting cycle, changing the root of T to j, and reversing the orientation of appropriate arcs in T to effect the rerooting. All the required operations, including determining the leaving arc (j, l), are performed using the original dynamic tree data structure of Sleator and Tarjan [20, 21], as spelled out in [22]. The time per pivot for these operations is $O(\log n)$.

Each execution of 1-4 takes a constant number of operations on the Euler tour structure and on the auxiliary heaps. (We leave the determination of exactly what operations are necessary as a routine exercise for the reader.) Thus the time required for one such execution is $O(\log n)$. Combining this with Orlin's bound of $O((nm) \min\{\log(nC), m \log n\})$ on the total number of executions of 1-4 gives us a bound of $O((nm \log n) \min\{\log(nC), m \log n\})$ on the total running time. Here n, m, and C are the number of vertices, number of edges, and maximum absolute value of an arc cost, respectively; the bound in terms of C requires integer arc costs. Our bound is better than Orlin's original bound by a factor of $O(n/\log n)$. As noted in the introduction, the same bound as ours can be obtained by using the original dynamic tree data structure extended as described in [10], but this structure is significantly more complicated than the Euler tour structure.

References

[1] G. M. Adel'son-Vel'skii and E. M. Landis, "An algorithm for the organization of information," *Soviet Math. Dokl.* **3** (1962), 1259–1262.

[2] R. Bayer and E. McCreight, "Organization of large ordered indexes," *Acta Inform.* **1** (1972), 173–189.

[3] S. Bent, D. Sleator, and R. E. Tarjan, "Biased search trees," *SIAM J. Computing* **14** (1985), 545–568.

[4] R. F. Cohen and R. Tamassia. "Dynamic expression trees and their applications," *Proc. 2nd ACM-SIAM Symposium on Discrete Algorithms* (1991), 52–61.

[5] J. Edmonds and R. M. Karp, "Theoretical improvements in algorithmic efficiency for netowrk flow problems," *J. Assoc. Comput. Mach.* **19** (1972), 248–264.

[6] D. Eppstein, G. F. Italiano, R. Tamassia, R. E. Tarjan, J. Westbrook, M. Yung, "Maintenance of a minimum spanning forest in a dynamic planar graph," *Proc. 1st ACM-SIAM Symp. on Discrete Algorithms* (1990), 1–11.

[7] G. N. Frederickson, "Data structures for on-line updating of minimum spanning trees," *SIAM J. Comput.* **14** (1985), 781–798.

[8] G. N. Frederickson, "Ambivalent data structures for dynamic 2-edge-connectivity and k smallest spanning trees," *Proc. 32nd IEEE Symp. on Foundations of Computer Science* (1991), 632–641.

[9] H. N. Gabow, Z. Galil, T. Spencer, and R. E. Tarjan, "Efficient algorithms for finding minimum spanning trees in undirected and directed graphs," *Combinatorica* **6** (1986), 109–122.

[10] A. V. Goldberg, M. D. Grigoriadis, and R. E. Tarjan. "Use of dynamic trees in a network simplex algorithm for the maximum flow problem," *Math. Prog.* **50** (1991), 277-290.

[11] A. V. Goldberg and R. E. Tarjan. "A new approach to the maximum flow problem," *J. Assoc. Comput. Mach.* **35** (1988), 921–940.

[12] A. V. Goldberg and R. E. Tarjan. "Finding minimum-cost circulations by canceling negative cycles," *J. Assoc. Comput. Mach.* **36** (1989), 873–886.

[13] A. V. Goldberg and R. E. Tarjan. "Finding minimum-cost circulations by successive approximation," *Math. of Oper. Res.* **15** (1990), 430–466.

[14] D. Goldfarb and J. Hao, "A primal simplex algorithm that solves the maximum flow problem in at most nm pivots and $O(n^2m)$ time," *Mathematical Programming* **47** (1990), 353–365.

[15] L. J. Guibas and R. Sedgewick, "A dichromatic framework for balanced trees," *Proc. Nineteenth Annual IEEE Symposium on Foundations of Computer Science* (1978), 8–21.

[16] M. R. Henzinger and V. King, "Randomized dynamic graph algorithms with polylogarithmic time per operation," *Proc. 27th Annual ACM Symp. on Theory of Computing* (1995), 519–527.

[17] G. L. Miller and J. H. Reif, "Parallel tree contraction and its application," *Proc. 26th Annual IEEE Symp. on Foundations of Comp. Sci.* (1985), 478–489.

[18] P. B. Miltersen, S. Subramanian, J. S. Vitter, and R. Tamassia. "Complexity models for incremental computation," *Theoretical Computer Science* **130** (1994), 203–236.

[19] J. B. Orlin. "A polynomial time primal network simplex algorithm," *Proc. 7th ACM-SIAM Symp. on Discrete Algorithms* (1996), to appear.

[20] D. Sleator and R. E. Tarjan. "A data structure for dynamic trees," *J. Computer and System Sciences* **26** (1983), 362–391.

[21] D. Sleator and R. E. Tarjan. "Self-adjusting binary search trees," *J. Assoc. Comput. Mach.* **32** (1985), 652–686.

[22] R. E. Tarjan. "Efficiency of the primal network simplex algorithm for the minimum-cost circulation problem," *Math. of Oper. Res.* **16** (1991), 272–291.

[23] R. E. Tarjan. "Updating a balanced search tree in $O(1)$ rotations," *Information Processing Letters* **16** (1983), 253–257.

[24] R. E. Tarjan. *Data Structures and Network Algorithms, CBMS 44,* Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.

[25] R. E. Tarjan and U. Vishkin. "An efficient parallel biconnectivity algorithm," *SIAM J. Comput.* **14** (1985), 862–874.