Implementation of Dynamic Trees with In-Subtree Operations

Tomasz Radzik

Department of Computer Science, King's College London

Technical Report, March 1998

ABSTRACT

We describe an implementation of dynamic trees with "in-subtree" operations. Our implementation follows Sleator and Tarjan's framework of dynamic-tree implementations based on splay trees. We consider the following two examples of "in-subtree" operations. (a) For a given node v, find a node with the minimum key in the subtree rooted at v. (b) For a given node v, find a random node with key X in the subtree rooted at v (value X is fixed throughout the whole computation). The first operation may provide support for edge deletions in the dynamic minimum spanning tree problem. The second one may be useful in local search methods for degree-constrained minimum spanning tree problems. We conducted experiments with our dynamic-tree implementation within these two contexts, and the results suggest that this implementation may lead to considerably faster codes than straightforward approaches do.

1. INTRODUCTION

A *dynamic tree collection* (or simply *dynamic trees*) is a data structure which maintains a collection of node-disjoint rooted trees under a sequence of operations including the following structural updates.

make_tree(k):	Create a new node with key k. The created node forms a new single-node tree.
link(v, w):	Add an edge from node v to node w to join two trees into one. Precondition: nodes v and w are in different trees and v is the root of its tree.
cut(v):	Delete the edge from node v to its parent. This operation splits one tree into two trees, if node v has not been the root of its tree.
evert(v):	Make node v the root of its tree.

The applications which use dynamic trees usually require the following queries about the current structure of the trees.

find_root(v):	Find the root of the tree containing node v.

Address: Department of Computer Science, King's College London, Strand, London WC2R 2LS, UK. Tel: +44-(0)171-873-2588. Fax: +44-(0)171-873-2851. E-mail: `radzik@dcs.kcl.ac.uk`

The preliminary version of this paper was titled "Implementations of dynamic tree collections based on splay trees" and appeared in the *Proceedings of the Workshop on Algorithm Engineering, WAE '97*, September 11-13, 1997, `http://www.dsi.unive.it/~wae97/proceedings`.

parent(v): Find the parent of node v.

For each node v, the parent of v, the children of v, and the subtree rooted at v are always defined with respect to the current root of the tree containing node v.

Each node v has a key, denoted by key(v), and different choices of operations on the node keys define different variants of dynamic trees. Sleator and Tarjan [1985] assume that node keys are real numbers and consider the following operations.

find_key(v): Return the value of the key of node v.

min_on_path(v): Find a node with the minimum key on the path from node v to the root.

add_key(v, x): Add number x to the key of every node on the path from node v to the root.

Operations like min_on_path and add_key refer to the path from a given node to the root, so we call them "on-path" operations.

Sleator and Tarjan [1985] show an implementation of the dynamic trees with the above two "on-path" operations (see also Tarjan [1983]). Their data structure is based on self-adjusting binary trees called *splay trees*, and requires $O(\log n)$ *amortized* time per one dynamic-tree operation. We denote by n the maximum number of nodes in one tree over a given sequence of dynamic-tree operations. The dynamic trees with operations min_on_path and add_key find applications in various network optimisation problems. For example, it has been shown that Sleator and Tarjan's implementation of such dynamic trees improves asymptotic bounds on the running times of many old as well as newly proposed network flow algorithms [Ahuja et al. 1989; Goldberg et al. 1990; Goldberg and Tarjan 1989; Goldberg and Tarjan 1990; Sleator and Tarjan 1983]. Their data structure can also be used in the dynamic minimum spanning tree problem to support insertions of new edges in $O(\log n)$ amortized time per one insertion [Frederickson 1985]. It should be noted that network flow algorithms and edge insertions in dynamic minimum spanning tree algorithms actually require dynamic trees with keys associated with edges not with nodes. However, in the context of such algorithms, the problem of maintaining edge keys can be easily reduced, without affecting asymptotic bounds on the running times, to the problem of maintaining node keys. Sleator and Tarjan [1983] describe an implementation of the dynamic trees with "on-path" operations which is based on biased search trees [Bent et al. 1985] and requires $O(\log n)$ *worst-case* time per one dynamic-tree operation, but has a practical disadvantage of being considerably more complicated than the data structure based on splay trees.

In this paper we consider dynamic trees in which the queries involving node keys refer to subtrees rather than to paths. We consider the following three types of such queries.

find_random_X(v):

Node keys are drawn from an arbitrary set. One key value, called value X, is distinguished. Operation find_random_X(v) selects a random node with key X in the subtree rooted at node v.

min_in_subtree1(v):

Node keys are real numbers. Operation min_in_subtree1(v) finds a node with the minimum key in the subtree rooted at node v.

min_in_subtree2(v, A):

Each node has q real-number keys. Number q does not have to be a constant, but is fixed for the whole computation. The i-th key of a node v is denoted by key(v, i). Operation min_in_subtree2(v, A), where v is a node and A is an array of size q, finds a node $x_{\min}$ and an index $i_{\min}$ which minimise key(x, i) over all nodes x in the subtree rooted at node v and all indices i such that $A[i] = 1$. That is, array A contains the characteristic vector of the set of indices of keys which are to be considered.

The applications of dynamic trees with "in-subtree" queries considered in this paper require also operation new_key, which changes a given node key to a given value. We describe an implementation of dynamic trees which gives $O(\log n)$ amortized time per one operation, if operation find_random_X or min_in_subtree1 is supported, and $O(q \log n)$ amortized time per one operation, if operation min_in_subtree2 is supported. Our implementation follows Sleator and Tarjan's implementation of dynamic trees with "on-path" operations based on splay trees. The added difficulty of implementing "in-subtree" operations as opposed to "on-path" operations is that in order to search efficiently a tree down towards the leaves, we need some mechanism of selecting the appropriate child from the possibly many children of the currently considered node. On the other hand, searching the path from a given node to the root seems to be a more straightforward process since each node has at most one parent. In Section 2 we describe details of Sleator and Tarjan's framework for implementing dynamic trees using splay trees, and in Section 3 we show how one can incorporate "in-subtree" operations into this framework.

Dynamic trees which support operation find_random_X can be used in implementations of local-search methods for constrained minimum spanning tree problems. Consider, for example, the NP-hard problem of finding a minimum-weight degree-3 spanning tree in a given weighted complete graph. A degree-3 tree is a tree such that each node has degree at most 3. If we apply the simulated-annealing method [Cerny 1985; Kirkpatrick et al. 1983] to this problem, we may have to maintain a degree-3 spanning tree under a sequence of operations of the following type. Remove a random edge from the current degree-3 spanning tree and reconnect the two parts into a new degree-3 spanning tree by adding a random edge which links two nodes of degree at most 2. Let the current degree-3 spanning tree T be rooted at an arbitrary node, and let each node v have either key X or key not_X, depending whether the degree of v in T is less than 3 (the node is available) or equal to 3 (the node cannot accept any additional tree edge). The described operation of exchanging random edges can be implemented by a constant number of dynamic-tree operations including operation find_random_X. In Section 4 we discuss details of such an implementation and its actual performance.

Dynamic trees with operations min_in_subtree1 and min_in_subtree2 can be used in the dynamic minimum spanning tree problem to support deletions of tree edges (that is, edges which are in the current minimum spanning tree). Let G denote the underlying dynamic undirected weighted graph, and let T denote the current minimum spanning tree of G (assume that G is always connected). For each node v in G, let T_v be a copy of tree T. The key of a node x in a tree T_v is equal to the weight of edge (v, x) in graph G, or to infinity, if (v, x) is not an edge in G. Each tree T_v is rooted at an arbitrary node. If a tree edge is deleted from the graph, tree T breaks into two trees T' and T'', and we have to find a replacement edge which reconnects these two trees back into a minimum spanning tree. Each node v contributes one candidate for the replacement edge: a minimum-weight edge which is adjacent to node v and joins trees T' and T''. Such an edge can be found by applying to tree T_v a constant number of dynamic-tree operations, including operation min_in_subtree1. Following this approach, we implement deletions of tree edges so that one deletion takes $O(n \log n)$ amortized time. Our experiments suggest that for dense graphs such an implementation can be quite competitive in practise. As discussed in Section 5, methods with considerably better asymptotic bounds have already been known for some time. However, consistently fast codes based on those asymptotically fast methods are yet to be developed. The existing codes sometimes perform substantially worse than even the ad-hoc method of re-computing the minimum spanning tree from scratch whenever a tree edge is deleted (see the experimental results reported by Amato, Cattaneo, and Italiano [1997]). In Section 5 we describe in detail how one can apply operations min_in_subtree1 and min_in_subtree2 to the dynamic minimum spanning tree problem, and we also present results from experiments with our dynamic minimum spanning tree codes.

All our codes are written in C++. Code mst_recompute is based on LEDA [Mehlhorn and Näher 1995]. Edge insertions in all our dynamic minimum spanning tree codes are supported by the implementation of Sleator and Tarjan's dynamic trees (with edge keys) developed by Sorin Moise. All experiments were conducted on Sun UltraSPARC (2×300 MHz, 512 MB). Our codes are available at http://www.dcs.kcl.ac.uk/staff/radzik.

2. GENERIC DYNAMIC TREES FOR IN-SUBTREE OPERATIONS

Our implementation of dynamic trees with "in-subtree" operations is based on the representation of dynamic trees by *virtual trees* proposed by Sleator and Tarjan [1985].

2.1 Virtual trees

A *virtual tree* D is a rooted tree such that for each node v in D, each child of v has one label from the set $\{$*left*, *right*, *dash*$\}$. A node cannot have more than one left and one right child, but may have a number of dash children. The left and the right children of a node v are called the *solid children* of v. Edges between nodes and their solid children are called *solid edges*, while the other edges are called *dash edges*. A maximal binary tree in D formed by solid edges is called a *solid tree*, and its root is called a *solid root*. For a node v of a virtual tree, solid_root(v) denotes the root of the solid tree containing v.

Each node v of a virtual tree has a *reverse bit* $b(v)$ associated with it. Let $B(v)$ denote the parity of the sum of the reverse bits on the path from node v to solid_root(v). A solid child u of node v is the *true left child* of v, if u is the left child of v and $B(v) = 0$, or u is the right child of v and $B(v) = 1$. Otherwise node u is

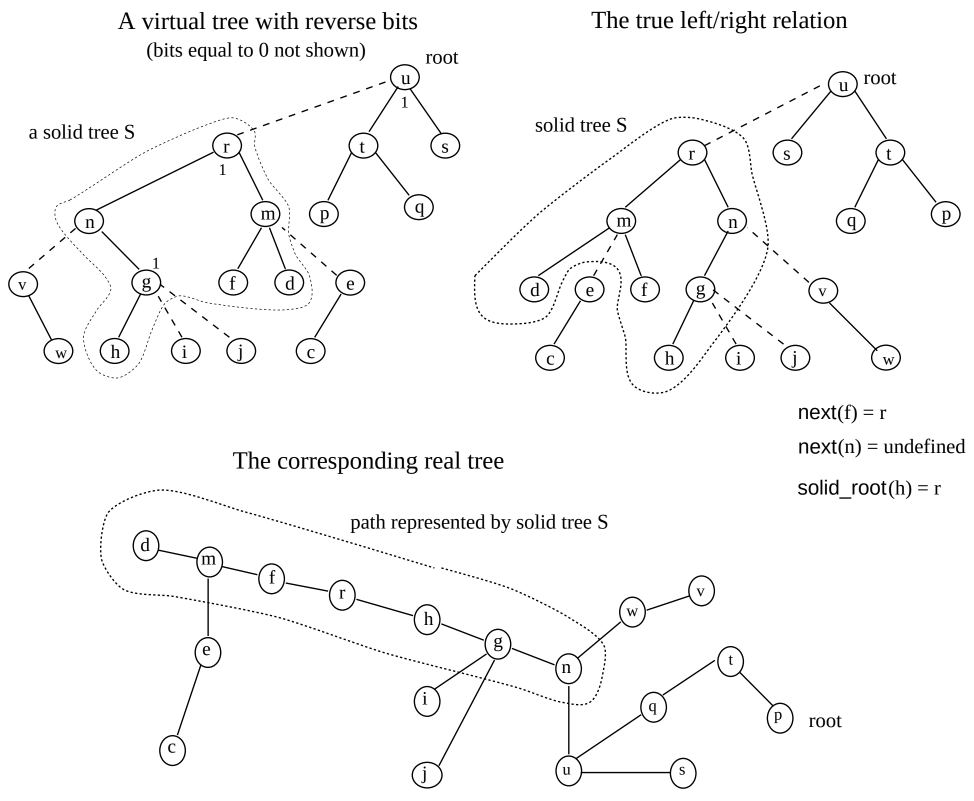

A virtual tree with reverse bits
(bits equal to 0 not shown)
root
a solid tree S
The true left/right relation
root
solid tree S
next(f) = r
next(n) = undefined
solid_root(h) = r
The corresponding real tree
path represented by solid tree S
root

Fig. 1. A virtual tree, the true left-right relation in this tree, and the corresponding real tree.

the *true right child* of v. The *true infix order* of the nodes of a solid tree is the infix order with respect to the true left-right relation. For a node v of a virtual tree D, let $\mathsf{next}_D(v)$ denote the successor of node v in the true infix order of the nodes of the solid tree containing node v.

Throughout this paper we use adjective "real" when we want to make it clear that we refer not to virtual trees but to dynamic trees which are represented by virtual trees. A virtual tree D represents a real tree T, if and only if, both trees have the same set of nodes and for each node v,

$$\mathsf{parent}_T(v) \;=\; \begin{cases} \mathsf{next}_D(v), & \text{if } \mathsf{next}_D(v) \text{ exists,} \\ \mathsf{parent}_D(\mathsf{solid\_root}_D(v)), & \text{if } \mathsf{next}_D(v) \text{ does not exists.} \end{cases}$$

The introduced terminology and the correspondence between a virtual tree D and the real tree T represented by D are illustrated in Figure 1. Observe that the solid trees in tree D correspond to some paths in tree T. Actually, the main concept of virtual trees is to provide a mechanism for representing long paths appearing in dynamic trees by binary trees with possibly small depths.

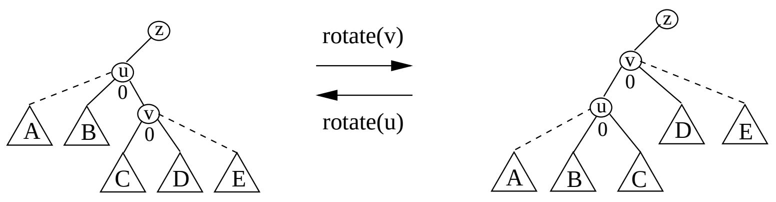


Fig. 2. Operation rotate in a virtual tree. "0" indicates that the reverse bit must be 0.

2.2 Primitive operations on virtual trees

We identify the following primitive operations on virtual trees: unreverse, rotate, splice, connect, disconnect, switch_bit, and make_tree. Operations unreverse, rotate, and splice have one argument, a node v, and alter locally the virtual tree D containing node v but do not affect the real tree represented by D. Operation unreverse(v) establishes at node v the locally correct left-right relation by performing the following update. If $b(v) = 1$, then $b(v)$ is set to 0, the solid children of v are swapped, and their reverse bits are reversed. Operation rotate(v) rearranges locally the subtree rooted at the solid parent u of node v as shown in Figure 2 (this is the rotation operation used in binary search trees). Node v moves one level up the tree and becomes the parent of node u. The precondition of this operation is that $b(v) = b(u) = 0$. Operation splice(v) can be applied when node v is a dash child of a solid root u and $b(u) = 0$. This operation swaps node v with its left sibling (see Figure 3).

Primitive virtual-tree operations connect, disconnect, and switch_bit affect the real trees. Operation connect($v, label, u$), where $label \in \{left, right, dash\}$, makes node v a child of node u as indicated by $label$. The required precondition is that nodes v and u are in different virtual trees, node v is the root of its virtual tree, and node u does not have the left (right) child, if $label = left$ ($label = right$). Operation disconnect(v) cuts the connection between node v and its parent. Operation switch_bit(v) changes the value of the reverse bit at node v to the opposite one (and, consequently, reverses the infix order of the nodes of the solid subtree rooted at v).

The virtual-tree operation make_tree creates a new single-node virtual tree. Since a single-node dynamic tree T is essentially the same as the single-node virtual tree D representing T, the dynamic-tree operation make_tree is simply the virtual-tree operation make_tree plus appropriate initialisation of the key-related attributes of the new node.

We do not assume that each primitive virtual-tree operation takes constant time. For example, operation rotate may involve some updates of key-related attributes, and these updates may require more than constant time.

2.3 Operation splay

Let v be a node in a virtual tree D, let u be the root of D, and let w be the root of the solid tree S containing node v. Operation solid_splay(v) rearranges tree S so that node v becomes the root of S. The precondition of this operation is that all reverse bits on the path in D from v to w are equal to 0. The rearrangement of tree S is accomplished by some sequence of operations rotate applied to nodes on the path from v to w. Operation solid_splay(v) is actually Sleator and Tarjan's splaying

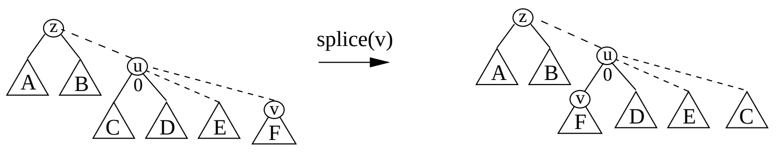


Fig. 3. Operation splice in a virtual tree. The reverse bit of node u must be 0.

operation in binary search trees [Sleator and Tarjan 1985] applied to node v of tree S.

Operation splay(v) brings node v to the root of the virtual tree by the following computation. First we apply operation unreverse to each node on the path from the root u to node v. Then we walk up the virtual tree from v to u performing operation solid_splay in each encountered solid tree. The path from node v to the (new) root z of the virtual tree consists now only of dash edges. Next we perform operation splice at each node on the path from v to z, and finally apply operation solid_splay to node v. Observe that operation splay uses only primitive virtual-tree operations unreverse, rotate, and splice, so it does not affect the real tree represented by virtual tree D. After operation splay(v) has been executed, the reverse bit of the new root (node v) is equal to 0. We use this condition in designing implementations of dynamic-tree operations.

Sleator and Tarjan [1985] show that any sequence of m virtual-tree operations splay, make_tree, connect, disconnect, and switch_bit, which is applied to the empty collection of virtual trees and includes n operations make_tree, requires $O(m \log n)$ primitive virtual-tree operations.

2.4 Implementation of basic dynamic tree operations

Operation splay is the main tool in implementations of dynamic-tree operations. This operation plays a dual role: It makes the crucial nodes roots of virtual trees, so necessary updates of the attributes of these nodes become straightforward; and it implicitly balances virtual trees, so the amortized running time of one dynamic-tree operation is low. Implementations of the basic dynamic-tree operations link, cut, and evert, are shown below.

```
cut($v$):      splay($v$);
               **if** $v$ has the right child $w$ **then** disconnect($w$)
               **else** $v$ is the root of its real tree (so there is nothing to cut).

link($v,w$):   splay($v$); splay($w$);
               **if** $w$ does not have the left child **then** connect($v, left, w$)
               **else** connect($v, dash, w$).

evert($v$):    splay($v$);
               **if** $v$ has the right child $w$ **then**
                   disconnect($w$); switch_bit($w$);
                   **if** $v$ does not have the left child **then** connect($w, left, v$)
                   **else** connect($w, dash, v$)
               **else** $v$ is the root of its real tree.
```

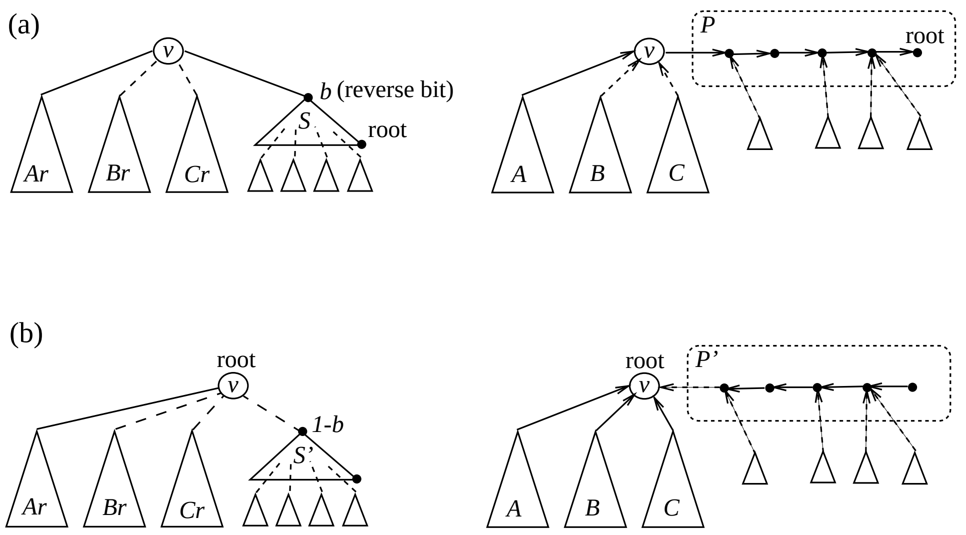


Fig. 4. Operation evert(v). (a) The virtual tree after operation splay(v) and the represented real tree (the arrows point towards the root). (b) The virtual tree and the represented real tree after the updates of operation evert(v). Trees Ar, Br, and Cr are virtual trees representing real trees A, B, and C.

Let us, for example, examine the implementation of operation evert(v). We assume that the edges of a real tree are directed towards the root. To make node v the root of its real tree, we have to reverse the directions of all edges on the path P from v to the root. Operation evert(v) begins with applying operation splay to node v. Node v becomes the root of its virtual tree and the solid right subtree S of node v represents path P (see Figure 4(a)). We reverse the directions of the edges on this path by switching the reverse bit at the root of subtree S and making S the left, if possible, or a dash subtree of v (see Figure 4(b)).

Query find_root(v) is implemented in the following way. We first apply operation splay to node v to bring v to the root of the virtual tree. The root r of the real tree is now the last node in the solid right subtree of v, according to the true infix order of nodes. Once we locate node r, we finish the computation by executing operation splay(r). Such “final” operations splay are required to maintain a desired bound on the amortized running time of one dynamic-tree operation.

```
find_root($v$):  splay($v$);  $r \leftarrow v$;
                 **while**  $r$ has the right child  **do**
                      $r \leftarrow$ the right child of $r$;  unreverse($r$);
                 splay($r$);  return $r$.
```

The implementation of query parent(v) is similar to the implementation of query find_root(v). The difference is that after operation splay(v), we search not for the last but for the first node in the solid right subtree of v.

A sequence of m dynamic-tree operations, which is applied to the empty collection of dynamic trees and includes n operations make_tree, is implemented by a sequence of $O(m)$ virtual-tree operations splay, make_tree, connect, disconnect, and switch_bit, which requires $t_1 = O(m \log n)$ primitive virtual-tree operations, plus some additional interstitial computation, which takes $t_2 = O(t_1)$ time. Thus the amortized time of one dynamic-tree operation is $O(d \log n)$, where d is a bound on the running time of one primitive virtual-tree operation.

3. IMPLEMENTATION OF "IN-SUBTREE" QUERIES

Let us consider the dynamic-tree operation min_in_subtree1(v). Each node w of a dynamic tree has a real number key(w) associated with it. We want to find a node with the minimum key in the subtree rooted at node v. To be able to locate quickly such a node, we maintain at each node w of a virtual tree the key key(w) and the minimum key min_key(w) in the virtual subtree rooted at w.

Operation min_in_subtree1(v) begins with executing operation splay(v). Node v is now the root of its virtual tree D, and the set of nodes of the real subtree rooted at v is the subset of nodes of the virtual tree D which consists of node v, the nodes of the left subtree of v, and the nodes of all dash subtrees of v. We navigate from the root v down the virtual tree D using values min_key. When we located a node x with the minimum key, we complete the computation by applying operation splay to node x (the "final" splay). Note that values min_key have to be appropriately updated during each primitive virtual-tree operation.

Let k denote the maximum degree of a node of a real tree. If the dash children of each node of a virtual tree are maintained simply in an unordered list, then one primitive virtual-tree operation may require $O(k)$ time. This would imply that the amortized time of one operation splay, and consequently the amortized time of one dynamic-tree operation, is $O(k \log n)$. To achieve an $O(\log n)$ bound, we use virtual trees to represent directly only such dynamic trees that each node has at most two children. We call such trees *restricted dynamic trees.* We implement *unrestricted dynamic trees* by representing them by restricted dynamic trees.

3.1 Restricted dynamic trees

We represent restricted dynamic trees by virtual trees such that each node has at most one dash child. Thus the information stored at a virtual node v consists of:

(a) four pointers, which point to the (virtual) parent, the left child, the right child, and the dash child of node v,

(b) the reverse bit of node v, and

(c) the information regarding the node keys.

The information regarding the node keys depends on the type of keys and the type of operations we want to supported. To support operation min_in_subtree1, each node v of a virtual tree contains key(v) and min_key(v), as described above. To support operation min_in_subtree2, each node v contains an array key of q keys of node v, and an array min_key of q minimum keys in the virtual subtree rooted at v: key(v, i) denotes the i-th key of node v and min_key(v, i) is equal to the minimum of key(x, i) over all nodes x in the virtual subtree rooted at v. To support operation find_random_X, we store at each node v the key key(v) of node v and the number of keys X in the virtual subtree rooted at v.

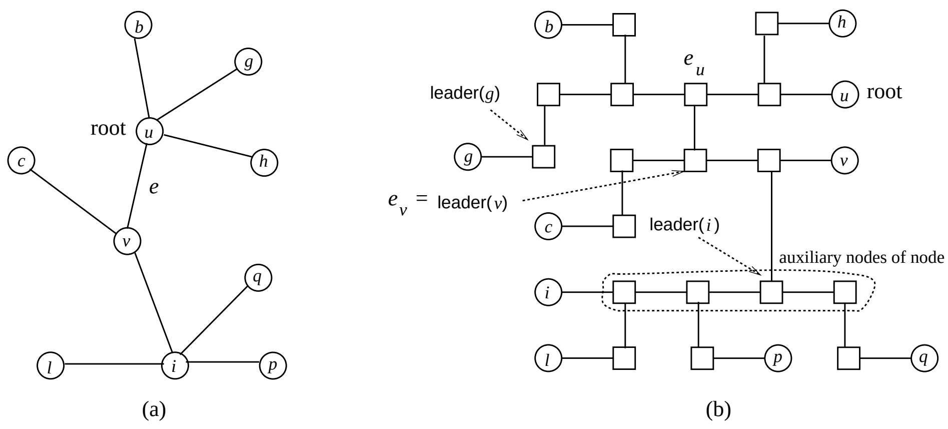


Fig. 5. (a) Unrestricted real tree T. (b) The corresponding restricted tree R.

In such virtual trees, each primitive virtual-tree operation runs in constant time, if we implement operation min_in_subtree1 or find_random_X, and in $O(q)$ time, if we implement operation min_in_subtree2. Thus in these two cases, the amortized time of one dynamic-tree operation on restricted trees is $O(\log n)$ and $O(q \log n)$, respectively.

3.2 Unrestricted dynamic trees

We represent unrestricted dynamic trees by restricted ones in the following way. Let T be an unrestricted dynamic tree. The restricted tree R representing tree T is essentially the adjacency-list structure of tree T (see Figure 5). Tree R has two types of nodes: the real nodes, that is, the nodes of tree T, and auxiliary nodes, which are used to represent the edges of tree T. If $e = (v, u)$ is an edge in tree T, then there are two auxiliary nodes e_v and e_u and edge (e_v, e_u) in tree R. The auxiliary node e_v is associated with the real node v. A real node v in tree R and all its auxiliary nodes are connected by edges to form a path; node v is an end node of this path. The root of tree R is the same as the root of tree T. If $e = (v, u)$ is an edge in tree T and u is the parent of v, then the auxiliary node e_v in tree R is called the *leader* of node v in tree R (see Figure 5). Observe that the real nodes of the subtree of tree R rooted at the leader of a real node v are exactly the nodes of the subtree of tree T rooted at v.

Some dynamic-tree operations on a collection C_1 of unrestricted trees, for example operations find_root and evert, are simply the same operations on the collection C_2 of restricted trees representing collection C_1 (that is, for example, operation evert(v) on C_1 is implemented by operation evert(v) on C_2). The other dynamic-tree operations on unrestricted trees can be implemented by a constant number of dynamic-tree operations on restricted trees. Actually, to achieve this we need two additional operations on restricted trees: child_r and leader. We use subscripts "r" and "u," when we want to make it clear whether we refer to restricted or unrestricted dynamic trees. For a node x of a restricted dynamic tree R, operation $\text{child}_\text{r}(x)$ returns an arbitrary child of node x in tree R. The implementation of this

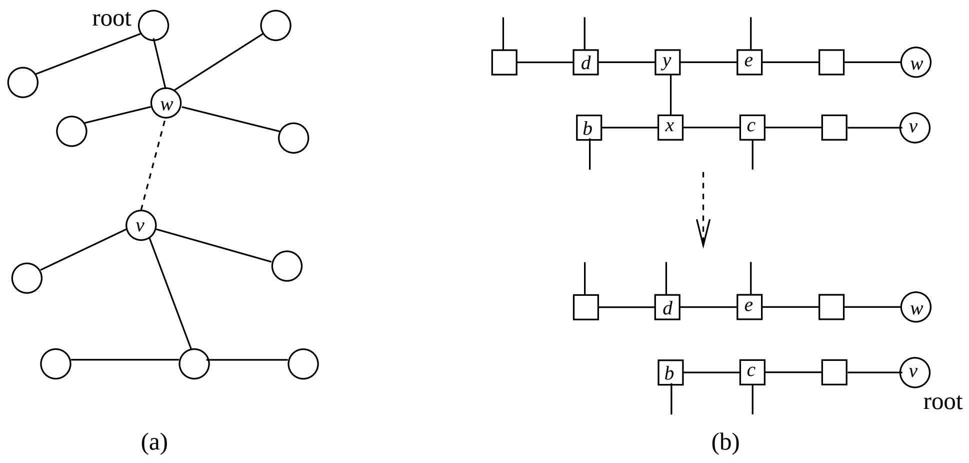


Fig. 6. (a) Operation $\mathsf{cut}(v)$ in an unrestricted tree. (b) The update of the corresponding restricted tree.

operation is based on the observation that after executing operation $\mathsf{splay}(x)$, the children of node x in tree R are the rightmost nodes in the left and the dash virtual subtrees of node x.

Operation $\mathsf{leader}(v)$ applies to a real node v of a restricted dynamic tree R representing an unrestricted dynamic tree, and returns the leader of node v in tree R. If node v is the root of tree R, then $\mathsf{leader}(v)$ returns v. To implement this operation, we maintain at each node x of tree R an attribute $\mathsf{real\_node}(x)$. If w is a real node in tree R and z is an auxiliary node of node w in tree R, then $\mathsf{real\_node}(z) = \mathsf{real\_node}(w) = w$. Let P be the path in tree R from node v to the root. The implementation of operation $\mathsf{leader}(v)$ is based on the observation that the nodes on path P with attribute $\mathsf{real\_node}$ equal to v form an initial segment of P, and the leader of node v is the last node of this segment. Thus to find the leader of node v, perform operation $\mathsf{splay}(v)$ and then find in the solid tree rooted at node v the last node x, in the true infix order, such that $\mathsf{real\_node}(x) = v$.

Operation parent on unrestricted dynamic trees has the following implementation.

$$\mathsf{parent}_{\mathrm{u}}(v): \quad \text{return } \ \mathsf{real\_node}(\mathsf{parent}_{\mathrm{r}}(\mathsf{leader}(v))).$$

To implement the “in-subtree” operations, we set the keys of auxiliary nodes in such a way that they do not interfere with the keys of the real nodes. If we implement the minimum-in-subtree queries, we set the keys of auxiliary nodes to infinity. If we implement operation $\mathsf{find\_random\_X}$, we set the keys of auxiliary nodes to any value different than X. Thus operation $\mathsf{min\_in\_subtree1}_{\mathrm{u}}(v)$ can be defined in the following way.

$$\mathsf{min\_in\_subtree1}_{\mathrm{u}}(v): \quad \text{return } \ \mathsf{min\_in\_subtree1}_{\mathrm{r}}(\mathsf{leader}(v)).$$

Operations $\mathsf{min\_in\_subtree2}_{\mathrm{u}}$ and $\mathsf{find\_random\_X}_{\mathrm{u}}$ have analogous definitions.

The implementations of operations link and cut on unrestricted dynamic trees are somewhat more involved. We give here only a detailed description of the implementation of operation cut. A sequence of dynamic-tree operations on restricted trees which implements operation cut(v) on unrestricted trees can be derived from Figure 6. Node x is leader(v) and node y is $\text{parent}_r(x)$. Nodes b, c, d, and e can be found using operations parent_r and child_r.

```
cut_u(v):
    x ← leader(v);
    if x = v then v is the root of its (unrestricted) tree
    else
        y ← parent_r(x); cut_r(x); evert_r(v);
        c ← parent_r(x); b ← child_r(x); d ← parent_r(y); e ← child_r(v);
        cut_r(x); if b exists then begin cut_r(b); link_r(b, c) end;
        cut_r(y); if e exists then begin cut_r(e); link_r(e, d) end;
        discard auxiliary nodes x and y.
```

The amortized time of one dynamic-tree operation on unrestricted trees is $O(\log n)$, if we implement operations min_in_subtree1 or find_random_X, and $O(q \log n)$, if we implement operation min_in_subtree2.

4. FLIP TREE

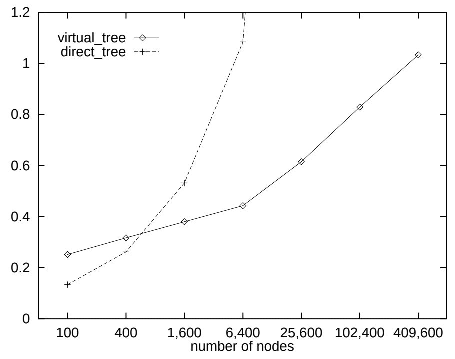
In this section we describe an application of dynamic trees with operation find_random_X. A degree-d *flip tree* is a data structure which maintains an n-node undirected degree-d tree T under a sequence of operations including operations of the following type.

random_flip:

> Remove a random edge from the current tree T, splitting T into two trees T_1 and T_2. Select two random nodes v and u of degree at most $d-1$, one from tree T_1 and the other from tree T_2. Using edge (v, u), join trees T_1 and T_2 into a new degree-d tree T'.

Such data structures arise in implementations of local search methods for degree-constrained minimum spanning tree problems. The notion of flip trees may be viewed as a generalisation of the *tour datatype*, defined and studied by Fredman, Johnson, McGeoch, and Ostheimer [1995]. The tour datatype maintains a traveling salesman tour and provides necessary operations for implementing local search methods for the traveling salesman problem. An "open" traveling salesman tour is a degree-2 spanning tree.

We implement a flip tree T by a collection of dynamic trees with operation find_random_X. We root tree T at an arbitrary node. The root keeps changing during the computation, but its exact position is never important outside of the data structure. Each node w of tree T has a key X or not_X, depending whether the degree of node w in tree T is less than d or equal to d. We need an additional dynamic-tree operation degree(w), which returns the (undirected) degree of a node w. Operation random_flip can be implemented in the following way.



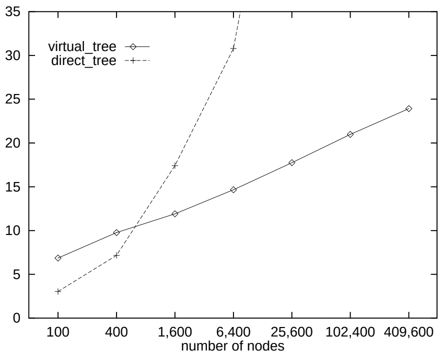
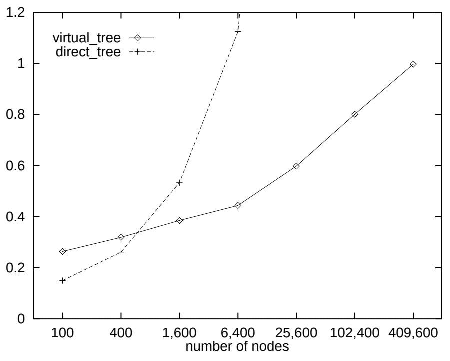


Fig. 7. A sequence of 10,000 operations random_flip, applied to an n-node tree obtained from "heap" by n operations random_flip. Left: Average (over 10 samples) execution times in seconds; for code direct_tree and the number of nodes $25,600$ and $102,400$, the times are 3.9 and 10.4, respectively. Right: Average depth of a node (for direct_tree the numbers are divided by 4).

```
random_flip:  $a \leftarrow$ a random node in $T$ other than the root;
              $b \leftarrow \mathsf{parent}(a)$;  $\mathsf{cut}(a)$;  $\mathsf{new\_key}(a, \mathsf{not\_X})$;  $\mathsf{new\_key}(b, \mathsf{not\_X})$;
              $v \leftarrow \mathsf{find\_random\_X}(a)$;  $\mathsf{evert}(b)$;  $u \leftarrow \mathsf{find\_random\_X}(b)$;
              $\mathsf{evert}(v)$;  $\mathsf{link}(v, u)$;
              **if** $\mathsf{degree}(v) = d$ **then** $\mathsf{new\_key}(v, \mathsf{X})$;
              **if** $\mathsf{degree}(u) = d$ **then** $\mathsf{new\_key}(u, \mathsf{X})$.
```

We have coded two degree-3 flip-tree data structures. Code virtual_tree is based on the implementation of restricted dynamic trees with operation find_random_X described in Sections 2 and 3. Code direct_tree is based on a straightforward implementation of restricted dynamic trees: A dynamic tree D is represented internally by a tree which has the same structure as D (that is, no splaying). In code virtual_tree the amortized running time of one operation random_flip is $O(\log n)$. In code direct_tree the average running time of one operation random_flip is linear in the average depth of a node in D.

We compare the performance of codes virtual_tree and direct_tree in the following experiment. We generate an n-node degree-3 tree and then perform a sequence of operations random_flip. The initial tree is obtained by applying a sequence of n operations random_flip to either the heap-like tree (node i is connected by edges with nodes $\lceil i/2 \rceil$, $2i$, and $2i+1$) or the path-like tree (node i is connected by edges



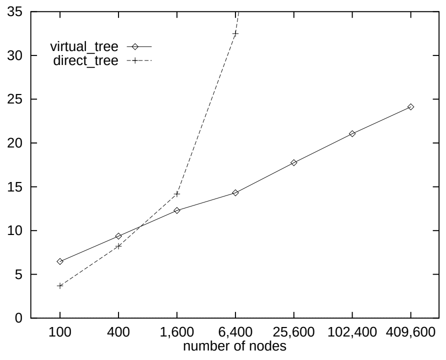


Fig. 8. A sequence of 10,000 operations random_flip, applied to an n-node tree obtained from "path" by n operations random_flip. Left: Average (over 10 samples) execution times in seconds; for code direct_tree and the number of nodes $25,600$ and $102,400$, the times are 3.4 and 8.7, respectively. Right: Average depth of a node (for direct_tree the numbers are divided by 4).

with nodes $i-1$ and $i+1$). In Figures 7 and 8 we plot average running times of sequences of 10,000 operations random_flip. We exponentially increase the number of nodes from 10^2 to $4 \cdot 10^5$. We also plot the average depth of a node in the tree representing the flip tree. These plots are consistent with the expectation that the average running time of one operation random_flip should be linearly related to the average depth of a node.

5. MINIMUM SPANNING FOREST IN DYNAMIC DENSE GRAPHS

Frederickson's topology-trees data structure [Frederickson 1985] together with the sparsification method [Eppstein et al. 1992] give a data structure for maintaining the minimum spanning forest in a dynamic graph in $O(\sqrt{n})$ worst-case time per one update – edge insertion or deletion. We denote by n and m the number of nodes and the number of edges in the underlying dynamic graph $G=(V,E)$. We assume that the set of nodes V is static, that is, it does not change during the computation. Recently Henzinger and King have improved this bound to $O(n^{1/3}\log n)$ amortized time per one update [Henzinger and King 1997]. Amato, Cattaneo, and Italiano [1997] conducted an extensive empirical study of performance of several methods for maintaining the minimum spanning forest, including various variants of Frederickson's data structure and the sparsification method, as well as the *ad hoc* method

code	edge insertions		edge deletions		random update (average)
	non-tree edge	tree edge	non-tree edge	tree edge	
mst1	Sleator-Tarjan dynamic trees		dynamic trees with operation min_in_subtree1		
	$O(\log n)$	$O(n \log n)$	$O(\log n)$	$O(n \log n)$	$\Theta\left(\frac{n^2}{m} \log n\right)$
mst2	Sleator-Tarjan dynamic trees		dynamic trees with operation min_in_subtree2		
	$O(\log n)$	$O(n \log n)$	$O(\log n)$	$O(n \log n)$	$\Theta\left(\frac{n^2}{m} \log n\right)$
mst_recompute	Sleator-Tarjan dynamic trees		recompute MST, if a tree edge deleted		
	$O(\log n)$	$O(\log n)$	$O(1)$	$O(m \log n)$	$\Theta(n \log n)$

Fig. 9. Asymptotic bounds for our dynamic minimum spanning forest codes.

of recomputing the whole minimum spanning forest whenever a tree edge, that is, an edge belonging to the minimum spanning forest, is deleted. Their experimental results show that their code adhoc, despite having $O(m \log n)$ worst-case and $O(m + n \log^2 n)$ average-case running time per one tree-edge deletion, outperforms their other codes, if the sequence of updates is random and the graph is not very sparse.

We have developed two codes, mst1 and mst2, for the dynamic minimum spanning forest problem which are based on our implementations of dynamic trees with operations min_in_subtree1 and min_in_subtree2, respectively. We have also developed a straightforward code mst_recompute which implements the *ad hoc* method. In all three codes insertions of edges are supported by the Sleator-Tarjan dynamic trees with operation min_on_path [Sleator and Tarjan 1985]. We denote this data structure by ST. In code mst_recompute, deletion of a tree edge is implemented by recomputing the whole minimum spanning forest using LEDA function MIN_SPANNING_TREE. The summary description of these three codes is shown in Figure 9. All three codes require $\Theta(n^2)$ memory and are intended for dense graphs. Further details of codes mst1 and mst2 are presented in Sections 5.1 and 5.2, respectively. In Section 5.3 we discuss an experiment which we have conducted to compare the actual performance of all three codes.

5.1 Code mst1: Deletions of edges supported by n dynamic tree collections

In code mst1, deletions of edges are supported by dynamic trees with operation min_in_subtree1 in the following way. Let F denote the current minimum spanning forest. For each node $x \in V$, we maintain a data structure $F(x)$, which is a dynamic collection of trees with operation min_in_subtree1 representing forest F. We maintain the invariant that the roots in all collections $F(x)$ are the same; that is, a given node is either a root in all collections $F(x)$ or in none of them. For each pair of nodes v and w, the key of node v in collection $F(w)$ and the key of node w in collection $F(v)$ are the same and are equal to the weight of edge (v, w), or to

```
delete_edge($v, u$):
    $C \leftarrow \emptyset$;    — the set of candidate edges for replacing edge $(v, u)$ —
    new_key($F(v), u, \infty$);  new_key($F(u), v, \infty$);
    if  $(v, u)$ is a tree edge  then
        delete edge $(v, u)$ from $ST$;
        — note: the directions of edges are the same in all collections $F(x)$, $x \in V$ —
        if  parent($F(v), u$) $= v$  then  begin  $temp \leftarrow v$;  $v \leftarrow u$;  $u \leftarrow temp$  end;
        $r \leftarrow$ find_root($F(v), u$);
        for  each $x \in V$  do
            cut($F(x), v$);
            — note: node find_root($F(x), u$) is node $r$ —
            if  find_root($F(x), x$) $= r$  then  $C \leftarrow C \cup \{(x,$ min_in_subtree1($F(x), v$)$)\}$
        if  $C \neq \emptyset$  then
            $(p, q) \leftarrow$ a minimum weight edge in $C$;  add edge $(p, q)$ to $ST$;
            for  each $x \in V$  do  begin  evert($F(x), p$);  link($F(x), p, q$);  end.
```

Fig. 10. Implementation of the edge-deletion operation in code mst1

infinity, if (v, w) is not an edge in the current graph.

The implementation of the edge-deletion operation is shown in Figure 10 (the details of the updates of data structure ST are omitted). For each dynamic-tree operation, we introduce an additional parameter indicating the dynamic-tree collection which is to be used. To delete an edge (v, u) from the graph, we first set to infinity the appropriate keys in collections $F(v)$ and $F(u)$ and check whether edge (v, u) is a tree edge. If (v, u) is a tree edge, then we perform the following updates. We delete edge (v, u) from data structure ST and find a replacement edge using collections $F(x)$, $x \in V$. Let the direction of edge (v, u) in collections $F(x)$, $x \in V$, be from v (child) to u (parent). The roots in all these collections are the same, so the directions of edges are also the same. For each node $x \in V$, we delete edge (v, u) from collection $F(x)$ by executing operation cut$(F(x), v)$. The removal of edge (v, u) from forest F splits one tree of F into two trees T_v and T_u. Tree T_v contains node v and tree T_u contains node u. We define a set of edges C in the following way,

$$C \; = \; \{(x, \mathsf{min\_in\_subtree1}(F(x), v)) \, : \, \mathsf{find\_root}(F(x), x) = \mathsf{find\_root}(F(x), u)\}.$$

Thus set C contains a minimum-weight edge joining node x and tree T_v, for each node x in tree T_u. A minimum-weight edge (p, q) from C replaces edge (v, u) in forest F. Using operations evert and link, we add edge (p, q) to $F(x)$, for each $x \in V$, and to ST.

Each insertion and deletion of a non-tree edge is implemented by a constant number of dynamic tree operations, so it runs in $O(\log n)$ amortized time. Each insertion and deletion of a tree edge involves $\Theta(n)$ dynamic-tree operations (a constant number of dynamic-tree operations in ST and in each collection $F(x)$, $x \in V$) so it runs in $O(n \log n)$ amortized time.

5.2 Code mst2: Deletions of edges supported by one dynamic tree collection

In code mst1, deletions of edges are supported by n separate dynamic tree collections, so a substantial part of the run time is always spent on restructuring virtual

```
delete_edge($v, u$):
    new_key($DT, v, u, \infty$);  new_key($DT, u, v, \infty$);
    if  $(v, u)$ is a tree edge  then
        delete edge $(v, u)$ from $ST$;
        evert($DT, u$);  cut($DT, v$);
        $P$ – an array indexed by nodes;
        for  each $x \in V$  do
            if  find_root($ST, x$) $=$ find_root($ST, u$)  then  $P[x] = 1$  else  $P[x] = 0$;
        $(p, q) \leftarrow$ min_in_subtree2($DT, v, P$);
        if  key($DT, p, q$) $< \infty$  then
            evert($DT, p$);  link($DT, p, q$);  add $(p, q)$ to $ST$.
```

Fig. 11. Implementation of the edge-deletion operation in code mst2

trees. To cut down on this restructuring, we develop code mst2, in which deletions of edges are based on representing the current minimum spanning forest F by one dynamic tree collection with operation min_in_subtree2. Let DT denote this collection. Each node in DT has an array of n keys indexed by nodes. The value key(DT, x, y) of the key at node x corresponding to node y is equal to the weight of edge (x, y), or to infinity, if (x, y) is not an edge in the current graph.

The implementation of the deletion of a graph edge (v, u) is shown in Figure 11. We first set keys key(DT, v, u) and key(DT, u, v) to infinity, and check whether edge (v, u) is a tree edge. If (v, u) is a tree edge, then we perform the following updates. We delete edge (v, u) from ST and DT. Deleting edge (v, u) from DT is done by executing operations evert(DT, u) and cut(DT, v). Let T_v and T_u be the new trees in forest F obtained by the removal of edge (v, u); tree T_v is the one which contains node v. Let P be an array of size n indexed by the nodes in V such that for each node $x \in V$, $P[x] = 1$ if and only if x belongs to tree T_u. The tree in collection DT representing tree T_v is rooted at node v. Therefore, operation min_in_subtree2(DT, v, P) returns a pair (p, q) of nodes which minimises key(DT, p', q') over all nodes p' which belong to tree T_v and all nodes q' which belong to tree T_u. If the returned pair is an edge in the current graph, then this edge replaces edge (v, u) in the minimum spanning forest F.

One dynamic-tree operation on collection DT takes $O(n \log n)$ amortized time. Each insertion and deletion of a tree edge involves a constant number of dynamic-tree operations on collection DT, so it takes $O(n \log n)$ amortized time. Each insertion and deletion of a non-tree edge involves two operations new_key on collection DT. In the virtual-tree representation of dynamic trees, changing the value of a key at a node v is normally done by applying operation splay(v) and then performing necessary updates of the key-related attributes of node v. If we followed this approach, the amortize time of one operation new_key on collection DT, and consequently the amortized time of one update involving a non-tree edge, would be $O(n \log n)$. To achieve a $O(\log n)$ bound, we apply operation splay(v) in operation new_key only if the path in the virtual tree from node v to the root is short.

We implement operation new_key(DT, x, y, k) (set the key of node x with index y to value k) in the following way. We change the value of key(DT, x, y) to value k, and then follow the path in the virtual tree from node x to the root, updating values min_key(DT, z, y) at each encountered node z. If this path contains more than $\gamma \log n$ nodes, where γ is some fixed constant, then we apply operation splay(x). The running time of a cheap operation new_key (an operation without splaying) is $O(\log n)$, while the running time of an expensive operation new_key is $\Theta(\delta n)$, where δ is the length of the path traversed in the virtual tree; $\delta \geq \gamma \log n$. Next we show that the amortized time of one operation new_key is $O(\log n)$.

Consider a sequence σ of dynamic-tree operations on collection DT, assuming that the collection is initially empty. Let q_1 denote the number of expensive operations new_key in sequence σ, and let q_2 denote the number of operations in sequence σ other than operations new_key. The running time of each cheap operation new_key is $O(\log n)$, and the running time of each operation in sequence σ other than a cheap operation new_key, amortized over all such operations, is $O(n \log n)$ (see Section 3.2), that is, at most $\beta n \log n$, for some constant β. Select constant γ such that the actual time of one expensive operation new_key is at least $2\beta n \log n$. Thus the total time spent on the expensive operations new_key in sequence σ is at least $2q_1\beta n \log n$, but cannot be greater than $(q_1 + q_2)\beta n \log n$. Therefore $q_1 \leq q_2$, and we can distribute the cost of the expensive operations new_key over the operations other than operations new_key. This way a cheap operation new_key, an expensive operation new_key, and any other operation in sequence σ have amortized running times $O(\log n)$, 0, and $O(n \log n)$, respectively. Hence the amortized running time of one operation new_key on collection DT is $O(\log n)$, so the amortized running time of one update of a non-tree edge in code mst2 is $O(\log n)$.

5.3 Actual performance of our dynamic minimum spanning tree codes

We use our dynamic minimum spanning tree codes mst1, mst2, and mst_recompute in the following experiment. We generate a random undirected weighted graph $G = (V, E)$ with n nodes and $n(n-1)/4$ edges, and then maintain the spanning tree in G under a random sequence of deletions and insertions of edges. Each type of update (deletion or insertion) has the same probability and edges for deletion and insertion are selected uniformly at random. The weights of the inserted edges are also chosen at random.

The average running times of all three codes are shown in Figure 12. The theoretical analysis predicts that the average running time of one update in such experiment should be $\Theta(\log n)$ for codes mst1 and mst2, and $\Theta(n)$ for code mst_recompute. Our experimental results seem to be consistent with this prediction. Code mst2 is about 2.5-3 times faster than code mst1. This difference is consistent with the ratio of the average number of updates of the node pointers to the average number of updates of the node attributes min_key in one operation splay in code mst1, which is roughly equal to 1.5. In code mst2, the number of updates of the node pointers becomes insignificant, while the number of updates of attributes min_key remains more-less the same as in code mst1. Code mst_recompute is based on the same method as code adhoc developed by Amato, Cattaneo, and Italiano [1997]. Since code adhoc performed quite well in their experiments and our code mst2 performs clearly better than code mst_recompute in our experiments, code mst2 may actually be quite competitive in practice.

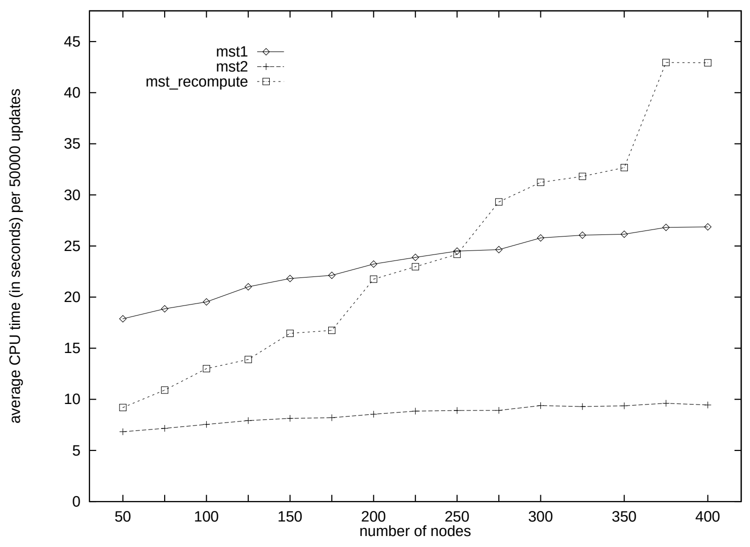


Fig. 12. Maintaining the minimum spanning tree in dynamic dense graphs. The initial density is 1/2. Edge weights are random numbers from $[0, 10000]$. The times are averages over 10 samples.

6. CONCLUSIONS AND FURTHER WORK

We have developed a generic implementation of dynamic trees which are to support "in-subtree" operations. Our implementation is based on Sleator and Tarjan's dynamic-trees data structure which uses splay trees [Sleator and Tarjan 1985]. Using our generic implementation, we implement three variants of dynamic trees, which support "in-subtree" operations find_random_X, min_in_subtree1, and min_in_subtree2, respectively. We use the first variant to implement a flip-tree data structure and the other two to implement two dynamic minimum spanning tree data structures. Our experimental results suggest that in some applications our dynamic-tree implementations may lead to substantially faster codes than codes based on more straightforward approaches.

One can think about a number of potential experiments which may provide further insight into practical value of our dynamic-trees implementations. For example, it would be interesting to evaluate the performance of our dynamic trees with operation find_random_X in the simulated-annealing method for degree-constrained minimum spanning tree problems. The next step in the evaluation of practical usefulness of our dynamic trees with operations min_in_subtree1 and min_in_subtree2 in the context of the dynamic minimum spanning tree problem could be a direct comparison of the performance of our codes mst1 and mst2 with the performance of codes which are based on theoretically fast methods.

Codes mst1 and mst2 are only practical for dense graphs. Could dynamic trees with operations min_in_subtree1 and min_in_subtree2 lead to data structures for the dynamic minimum spanning tree problem which are practical also for sparse graphs? It seems that to maintain efficiently the minimum spanning tree in a sparse graph, one needs to maintain a hierarchical decomposition of the tree. Observe that the virtual-tree representation of a dynamic tree T can be actually viewed as a hierarchical decomposition of T. We believe that this decomposition can be further exploited.

Our flip-tree data structure (code virtual_tree in Section 4) can be viewed as a generalisation of the dynamic traveling-salesman data structure based on splay trees. Fredman, Johnson, McGeoch, and Ostheimer [1995] describe this data structure and show that it gives good performance of local-search optimisation algorithms for TSP problem. They also show, however, that some other dynamic traveling-salesman data structures, for example their data structure based on two-level trees, are usually faster. It would be interesting to design and evaluate a generic dynamic-tree data structure which generalises the two-level-tree data structure.

REFERENCES

Ahuja, R. K., Orlin, J. B., and Tarjan, R. E. 1989. Improved Time Bounds for the Maximum Flow Problem. *SIAM J. Comput. 18*, 939–954.

Amato, G., Cattaneo, G., and Italiano, G. F. 1997. Experimental analysis of dynamic minimum spanning tree algorithms. In *Proc. 8th Annual ACM-SIAM Symposium on Discrete Algorithms* (1997).

Bent, S. W., Sleator, D. D., and Tarjan, R. E. 1985. Biased Search Trees. *SIAM J. Comput. 14*, 3, 545–568.

Cerny, V. 1985. A thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm. *J. Optimization Theory and Appl. 45*, 41–51.

Eppstein, D., Galil, Z., Italiano, G. F., and Nissenzweig, A. 1992. Sparsification – a technique for speeding up dynamic graph algorithms. In *Proc. 33rd IEEE Annual Symposium on Foundations of Computer Science* (1992), pp. 60–69.

Frederickson, G. N. 1985. Data structures for on-line updateing of minimum spanning trees, with applications. *SIAM J. Comput. 14*, 4, 781–798.

Fredman, M. L., Johnson, D. S., McGeoch, L. A., and Ostheimer, G. 1995. Data structures for traveling salesmen. *J. Alg. 18*, 3, 432–479.

Goldberg, A. V., Tardos, E., and Tarjan, R. E. 1990. Network Flow Algorithms. In B. Korte, L. Lovás, H. Prömel, and A. Schrijver Eds., *Paths, Flows, and VLSI-layout*, pp. 101–164. Springer-Verlag.

Goldberg, A. V. and Tarjan, R. E. 1989. Finding minimum-cost circulations by canceling negative cycles. *J. Assoc. Comput. Mach. 36*, 388–397.

Goldberg, A. V. and Tarjan, R. E. 1990. Finding Minimum-Cost Circulations by Successive Approximation. *Math. Oper. Res. 15*, 430–466.

Henzinger, M. R. and King, V. 1997. Maintaining minimum spanning trees in dynamic graphs. In *Proc. 24th International Colloquium on Automata, Languages and Programming* (July 1997).

Kirkpatrick, S., C. D. Gelatt, J., and Vecchi, M. P. 1983. Optimization by simulated annealing. *Science 220*, 671–680.

Mehlhorn, K. and Näher, S. 1995. LEDA, A platform for combinatorial and geometric computing. *Comm. ACM 38*, 1, 96–102.

Sleator, D. D. and Tarjan, R. E. 1983. A Data Structure for Dynamic Trees. *J. Comput. System Sci. 26*, 362–391.

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

Tarjan, R. E. 1983. *Data Structures and Network Algorithms.* Society for Industrial and Applied Mathematics, Philadelphia, PA.