Algorithmica (1992) 7: 433–464

Algorithmica

Maintaining Bridge-Connected and Biconnected Components On-Line[1]

Jeffery Westbrook[2] and Robert E. Tarjan[3]

Abstract. We consider the twin problems of maintaining the bridge-connected components and the biconnected components of a dynamic undirected graph. The allowed changes to the graph are vertex and edge insertions. We give an algorithm for each problem. With simple data structures, each algorithm runs in $O(n \log n + m)$ time, where n is the number of vertices and m is the number of operations. We develop a modified version of the dynamic trees of Sleator and Tarjan that is suitable for efficient recursive algorithms, and use it to reduce the running time of the algorithms for both problems to $O(m\alpha(m, n))$, where α is a functional inverse of Ackermann's function. This time bound is optimal. All of the algorithms use $O(n)$ space.

Key Words. On-line algorithms, Graph algorithms, Graph connectivity, Dynamic trees, Data structures.

1. Introduction. Three natural and important equivalence relations on the constituents of an undirected graph $G = (V, E)$ are defined by its connected, bridge-connected, and biconnected components. Let $V_1, V_2, \ldots, V_k$ be the partition of V such that two vertices are in the same part of the partition if and only if there is a path connecting them. Let G_i be the subgraph of G consisting of V_i and the edges of E incident to the vertices of V_i. The subgraphs G_i, $1 \leq i \leq k$, form the connected components, abbreviated *components*, of G. Similarly, let $E_1, E_2, \ldots, E_k$ be the partition of E such that two edges are in the same part of the partition if and only if they are contained in a common simple cycle. The biconnected components, or *blocks*, are the subgraphs of G formed by the edges of E_i and the vertices of V that are endpoints of these edges. A vertex appearing in more than one block is called an *articulation point*, and its removal disconnects G. There is either no block or one block containing any given pair of vertices. An edge contained in no cycle is in a block by itself. Such an edge is called a *bridge*, and its removal disconnects the graph. The bridge-connected components, or *bridge-blocks*, of G are the components of the graph formed by deleting all the bridges. The bridge-blocks partition V into equivalence classes such that two vertices are in the same class if and only if there is a (not necessarily simple) cycle of G containing both of them. Figure 1 shows an undirected graph along with its blocks and bridge-blocks.

[1] Research at Princeton University supported in part by National Science Foundation Grant DCR-86-05962 and Office of Naval Research Contract N00014-91-J-1463.
[2] Department of Computer Science, Yale University, P.O. Box 2158, Yale Station, New Haven, CT 06520-2158, USA. This work was partially done while the author was at the Department of Computer Science, Princeton University, Princeton, NJ 08544, USA.
[3] Department of Computer Science, Princeton University, Princeton, NJ 08544, USA, and NEC Research Institute, Princeton, NJ 08540, USA.

Received September 1, 1989; revised June 8, 1990. Communicated by Greg N. Frederickson.

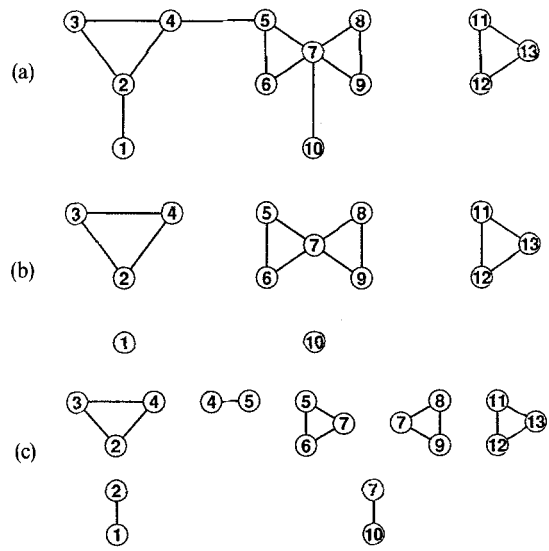

Fig. 1. (a) Undirected graph G. (b) Bridge-blocks of G. (c) Blocks of G. Multiply-appearing vertices are articulation points.

The problems of finding the components, blocks, and bridge-blocks of a fixed graph are well understood. Hopcroft and Tarjan [9] and Tarjan [18] give sequential algorithms that run in time $O(n + m)$, where $n = |V|$ and $m = |E|$. Logarithmic-time parallel algorithms for finding components, bridge-blocks, and blocks are given in [1] and [24] (see also the survey paper [10]).

The problem of answering queries about edge and vertex membership in the components of a dynamic graph, i.e., a graph that is changing on-line, has been addressed in [5]–[7] amd [13]. Even and Shiloach [6] consider the component problem for a graph undergoing edge deletions. They give algorithms with constant query time, $O(n \log n)$ total update time in the case that G is a tree or forest, and $O(mn)$ update time for general G, where m and n are the numbers of edges and vertices, respectively, in the initial graph. Reif [13] gives an algorithm for the same problem that runs in time $O(ng + n \log n)$ when given an initial graph embedded in a surface of genus g. Frederickson [7] gives an algorithm that performs queries in constant time and edge insertions and deletions in time $O(\sqrt{m_i})$, where m_i is the number of edges in G at the time of the ith update. It is well known (see, for example, [6] and [13]) that if only edge insertions are allowed, the component problem can be solved by straightforward application of a fast disjoint set union algorithm. The disjoint set union problem is to maintain a partition of n elements while performing an intermixed sequence of two operations: *find*(x), which returns the name of the set containing element x; and *union*(A, B), which combines the sets named A and B into a new set named A. The fastest algorithms for this problem run in $O(\alpha(m, n))$ amortized time per operation[4] and

[4] The amortized cost of an operation is the cost of a worst-case sequence of operations divided by the number of operations in the sequence. See [22] for a general discussion of amortization.

$O(n)$ space, where m is the length of the sequence, n is the total number of elements, and α is a functional inverse of Ackermann's function [19], [23].

In this paper we study the problems of answering queries about the blocks or bridge-blocks of a dynamic graph. We allow two on-line graph update operations to be performed on an initially null graph G:

make vertex(A): Add a new vertex with no incident edges to G. Label by "A" the bridge-block formed by the new vertex. Return the name of the new vertex; the calling program must use this name to refer to the new vertex in subsequent operations. (The name is actually a pointer into the data structure maintained by the algorithm.)

insert edge(u, v, A): Insert a new edge between the vertices named u and v. Label by "A" any new bridge-block or block that results from the edge insertion.

In the bridge-block problem we allow the following query:

find block(u): Return the label of the bridge-block containing the vertex named u.

Similarly, in the block problem we allow the following query:

find block(u): Return the label of the bridge-block containing the vertex named u. vertices $\{u, v\}$.

We also consider a restricted variant of the problem in which we are given an initially connected graph $G_0 = (V_0, E_0)$. We allow $O(|E_0|)$ preprocessing time, and then process on-line a sequence of intermixed queries and edge insertions. In this variant, the *make vertex* operation is not allowed. Our algorithms do not explicitly maintain the edge set, but if the endpoints of an edge are known, then *find block* can be used to determine which block contains the edge.

The block and bridge-block problems are fundamental problems in the study of on-line graph algorithms, a study that has wide applications to networks, CAD/CAM, and distributed computing. They appear as subproblems of other on-line graph problems, such as incremental planarity testing [2]. The incremental planarity testing problem is to maintain a representation of a planar graph as edges are being added, and to determine the first edge addition that makes the graph nonplanar. Maintaining the blocks of a dynamic graph also arises in the implementation of efficient search strategies for logic programming (Graeme Port, private communication, 1988). Tamassia [16], [17] gives a data structure for the on-line block problem that achieves $O(\log n)$ worst-case query time and $O(\log n)$ amortized update time.

This paper is organized as follows. In Section 2 we develop a simple algorithm for the bridge-block problem that runs in $O(n \log n + m)$ total time and uses $O(n)$ space, where n is the number of vertices added to the initially null graph and m is the number of operations. In Section 3 we give a similar algorithm for the block problem that also runs in $O(n \log n + m)$ time and $O(n)$ space. Both these data structures answer queries in $O(1)$ worst-case time. In Sections 4–6 we introduce the link/condense tree data structure, a modified version of the dynamic trees of Sleator and Tarjan [14], [15]. By a delicate analysis we show that this data

structure reduces the amortized time per operation to $O(\alpha(m, n))$. Link/condense trees support condensing of adjacent nodes, and two such trees can be linked together in amortized time $O(\log k)$, where k is the size of the smaller tree. Section 4 introduces the data structure and Sections 5 and 6 give details of its implementation and analysis. Finally, Section 7 contains a simple reduction from the disjoint set union problem to the block and bridge-block problems. This allows us to apply the known lower bounds for set union to these two problems. Our results are summarized in the following table:

	Initially connected G	Initially null G
Bridge-blocks	$O(m\alpha(m, n))$	$\Theta(m\alpha(m, n))$
Blocks	$\Theta(m\alpha(m, n))$	$\Theta(m\alpha(m, n))$

The $O(\alpha(m, n))$ upper bound for the on-line bridge-block and block problems is somewhat surprising, since both these problems differ fundamentally from the on-line connected component problem that shares this bound. In the latter a single edge insertion can combine at most two components into one. A single edge insertion can create a cycle in the graph that might combine as many as $\Theta(n)$ blocks or bridge-blocks into one, however. This fact seems to make the maintenance of blocks and bridge-blocks under both edge insertion and edge deletion quite difficult. By simply alternating edge insertions with edge deletions, we can generate a sequence of operations that at every step changes the number of blocks in the graph by $\Theta(n)$.

Working independently and using techniques different from ours, La Poutré *et al.* have recently also obtained an $O(m\alpha(m, n))$ time bound for maintaining bridge-blocks and for maintaining three-edge connected components [12]. Their results, as well as other recent work that builds on the preliminary version of the present paper [25] are briefly discussed in Section 8.

2. Maintaining Bridge-Blocks On-Line. The bridge-blocks and bridges of a connected graph have a natural tree structure that we call the *bridge-block tree*. The collection of bridge-block trees given by the components of a graph $G = (V, E)$ is called the *bridge-block forest*, or BBF. The nodes in a bridge-block tree are of two types: square nodes, which represent the vertices of G; and round nodes, which represent the bridge-blocks. If r is a round node, then *label*(r) denotes the label of the corresponding bridge-block. Two round nodes are connected by a tree edge whenever a bridge connects the corresponding two bridge-blocks. If vertex v is contained in bridge-block A, then the square node that represents v is connected by a tree edge to the round node that represents A. Every square node is a leaf. Each bridge-block tree is rooted at an arbitrarily selected round node. We denote by *parent*(x) the parent of tree node x. Given the BBF for G, the query *find block*(u) can be answered by computing *label*(*parent*(u)). (In general, we let the vertex name, here u, refer to both the vertex in the graph and to its square node

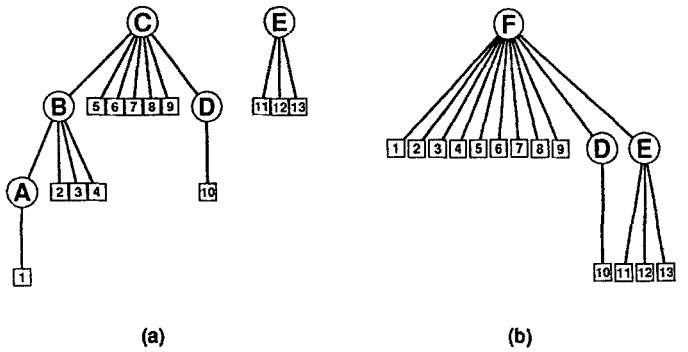

Fig. 2. (a) BBF of Figure 1. (b) BBF after performing *insert edge*(1, 5, *F*) and *insert edge*(9, 11, *E*).

representative in the bridge-block tree.) Figure 2(a) shows the BBF for the graph of Figure 1.

The effect of an *insert edge*(u, v, A) operation on the BBF depends on whether u and v are in the same bridge-block, different bridge-blocks but the same component, or different components. In the first case the bridge-block structure of the graph, and consequently the BBF, is unaffected. In the remaining two cases, however, the bridge-block structure is affected in opposite ways.

If u and v are in different components of the graph, then the inserted edge connects the two components by creating a bridge between them. One of the two corresponding bridge-block trees, say the tree containing u, is rerooted at *parent*(u), and then *parent*(u) is made a child of *parent*(v). This process, called a *component link*, can occur at most $n - 1$ times, after which the graph is connected.

On the other hand, if u and v are in the same component but different bridge-blocks, the inserted edge creates a new cycle that reduces the number of bridge-blocks in the component. The round nodes on the path connecting u and v in the corresponding tree must be replaced by a single round node. Every node previously adjacent to one of the round nodes on the path becomes adjacent to the new single round node. This process is called *path condensation*. After at most $n - 1$ condensations, the graph has only a single bridge-block. Figure 2(b) gives examples of both cases of edge insertion.

To implement the bridge-block operations, the BBF is explicitly maintained with a data structure that supports the following functions.

maketree(A: label or null value): Create a new tree consisting of a single node. If A is null, make a square node; otherwise, make a round node labeled A. Return a pointer to the new node.

evert(x: square or round node): Make x the root of the tree that contains x (this is called an *eversion*).

link(x, y: square or round nodes): Link two trees together by making x, assumed to be the root of its tree, a child of y.

findpath(u, v: square nodes): Return the tree path P between but not including square nodes u and v. If u and v are the same node, return *parent*(u).

condensepath(P: path; A: label): Perform path condensation on P and label by A the resultant single node $\tilde{r}$.

With these functions, we implement the bridge-block operations as follows:

make vertex(A): Let u = *maketree*(null). Perform *link*(u, *maketree*(A)). Return u (to be used by the calling program as the name of the new vertex).
find block(u): Return *label*(*findpath*(u, u)).
insert edge(u, v, A): If u and v are in different components denoted C_u and C_v, respectively, let x = *findpath*(u, u) and y = *findpath*(v, v). If $|C_u| \leq |C_v|$, execute *evert*(u) and *link*(u, v); otherwise execute *evert*(v) and *link*(v, u). If u and v are in the same component, execute *condensepath*(*findpath*(u, v), A).

An on-line component maintenance subroutine is used in the *insert edge* procedure to determine if two vertices are in the same component of G and to determine the size of a component. This subroutine is a straightforward application of a fast disjoint set union algorithm [23]. Appropriate calls to the update functions of this subroutine must be made when making a new vertex or performing a component link.

The tree data structure is built using *condensible nodes*. A condensible node x consists of a block of storage, $N(x)$, containing an arbitrary but fixed collection of fields, and a set of subnodes, $S(x)$. The subnode sets are maintained with a fast disjoint set union algorithm [23]. The name of set $S(x)$ is simply $N(x)$. A condensible node is initialized with one subnode. To make a pointer p to node x, we store in p the name of some subnode $s \in S(x)$. Given such a pointer p, a *pointer step* consists of accessing $N(x)$ by performing *find*(p). Two nodes x and y can be condensed into a single node z by the following procedure: create a new storage block $N(z)$; let $S(z)$ = *union*($S(x)$, $S(y)$); update appropriately the fields of $N(z)$ using the fields of $N(x)$ and $N(y)$; and discard $N(x)$ and $N(y)$. The union of the two subnode sets suffices to make all pointers to x and y become pointers to z. Note that condensation destroys x and y. If a data structure initially contains n condensible nodes, then any sequence of m pointer steps and condensations runs in worst-case time $O(m\alpha(m, n))$, and the data structure uses $O(n)$ space.

For each vertex v in the graph we store a pointer to its condensible node representative in the data structure. For a node x, $N(x)$ contains a parent pointer and label field. If x is a square node, the label field is null; if x is a round node, the label field holds a bridge-block label. For any node x, *parent*(x) is computed by a pointer step using the parent pointer. The four tree functions take as arguments pointers to condensible nodes.

An eversion at x is done by walking up from x to the tree root, reversing the direction of all parent pointers along the path. The number of pointer steps is the initial depth of x, which is at most the size of the tree. To perform *link*(x, y) a pointer to y is stored in the parent field of x. Since x and y are passed in the form of pointers, this is actually implemented by storing the value of y in the parent field of the storage block returned by *find*(x). This requires one additional pointer step.

The *findpath*(u, v) algorithm proceeds by walking up the tree simultaneously

from u and from v in lock-step, until the paths from u and v intersect at their nearest common ancestor. The path P is returned as a list of nodes (not in order along the path), with the nearest common ancestor at the end. The number of pointer steps required is at most $2|P|$. Let z be the parent of the nearest common ancestor. To perform *condensepath*(P, A), P is condensed into a single node $\tilde{r}$. Node $\tilde{r}$ is labeled by A and made a child of z. The number of node condensations is $|P| - 1$.

LEMMA 1. *In any sequence of operations there are $O(n)$ pointer steps during condensing edge insertions.*

PROOF. Suppose a path P is being condensed. To generate P, $O(|P|)$ pointer steps are required, but $|P| - 1$ round nodes are condensed. After $n - 1$ condensations the graph is bridge-connected. □

LEMMA 2. *The total number of pointer steps in evert and link operations is $O(n \log n)$.*

PROOF. The number of square nodes in a bridge-block tree is at least the number of round nodes. Hence the cost of an evert in a tree of k square nodes is $\Theta(k)$. Since we always evert the smaller tree, we arrive at the following recursive upper bound on the total number of pointer steps, $T(n)$, needed to combine n components into one, where c is a sufficiently large constant:

$$T(1) = 0,$$

$$T(n) \leq \max_{1 \leq k \leq n/2} \{T(k) + T(n - k) + ck\}.$$

It is well-known that this recurrence has the solution $T(n) = O(n \log n)$. □

THEOREM 1. *Given an initially null graph G_0, the above data structure processes a sequence of $m = \Omega(n)$* find block, make vertex, *and* insert edge *operations in $O(n \log n + m)$ time and $O(n)$ space, where n is the number of vertices inserted.*

PROOF. Let k be the total number of pointer steps and node condensations performed while processing the request sequence. There are a constant number of pointer steps per find block operation. Together with Lemmas 1 and 2, this implies that $k = O(n \log n + m)$. The amortized cost of a condensible node operation is $O(\alpha(k, n))$. Thus the total running time is $O(k\alpha(k, n))$. Since $\alpha(a, b) = 1$ for $a \geq b \log \log b$ [21], this expression is $O(n \log n + m)$. The total time spent in the auxiliary on-line components subroutine is $O(m\alpha(m, n))$, which is also $O(n \log n + m)$. The space bound follows from the properties of condensible nodes and the observation that the number of square nodes bounds the number of round nodes. □

COROLLARY 1. *Given an initially connected graph G_0 and $O(|E_0|)$ preprocessing time, a sequence of $m = \Omega(n)$* find block *and* insert edge *operations can be processed in $O(m\alpha(m, n))$ time.*

PROOF. The bridge-blocks and initial BBF of G_0 can be found in time $O(|E_0|)$ using one of the algorithms in [9] and [18]. By Lemma 1, the total number of pointer steps and condensations is $O(m)$, giving the bound. □

3. Maintaining Blocks On-Line. The problem of maintaining blocks on-line is similar to that of maintaining bridge-blocks, and the algorithms are almost identical. We represent each block of G by a round node and each vertex by a square node. The square nodes now play a more important role, however, and the *block forest*, or BF, is different in character from the BBF. Whenever a vertex of G belongs to a given block, we create a tree edge between the corresponding square and round nodes in the BF. There are no other edges; no two square nodes and no two round nodes are adjacent. Since a single vertex can be an articulation point joining many blocks, and a block may be adjacent to many articulation points, the block tree generally has internal square nodes. If $\{u, v\}$ is a vertex pair that appears in block B, then in the block tree either u and v are both children of B or one is a parent of B and one is a child of B. The query *find block*(u, v) can be answered by returning the label of the single round node that lies on the tree path between u and v. Figure 3(a) gives the BF for the graph of Figure 1.

A *make vertex*(A) operation adds a single unconnected square node to the BF. An *insert edge*(u, v, A) operation can have two opposite effects: either it links two components, increasing the number of blocks, or it creates a new cycle, possibly decreasing the number of blocks. If a component link occurs, then the inserted bridge forms a new block labeled by A. One of the two block trees, say the tree containing u, is rerooted at u; u becomes a child of A, and A becomes a child of v.

If the inserted edge creates a new cycle, then path condensation occurs. In a block tree the path between u and v consists of alternating square and round nodes. The round nodes on the path are condensed into a single round node, $\tilde{r}$. The square nodes on the path may be ignored, since condensation of the round nodes ensures that these square nodes become children of $\tilde{r}$. Let w be the nearest common ancestor of the path nodes. (Recall that this implies that w is part of the

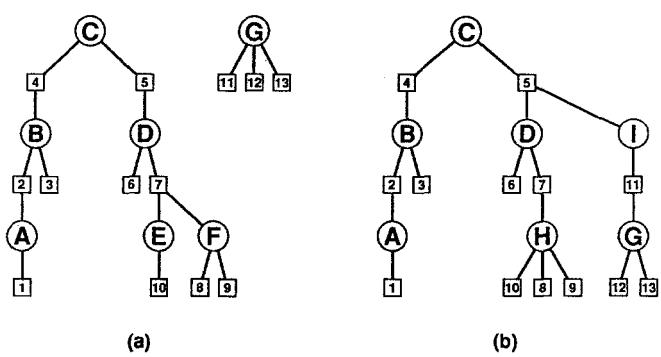

Fig. 3. (a) BF of the graph of Figure 1. (b) BF after performing *insert edge*$(8, 10, H)$ and *insert edge*$(5, 11, I)$.

path.) If w is a round node, then $\tilde{r}$ becomes a child of the parent of w. If w is a square node, it is not affected by condensation, and $\tilde{r}$ becomes a child of w. Examples of a linking edge insertion and a condensing edge insertion with a square nearest common ancestor are given in Figure 3(b).

The BF is maintained with the same condensible node data structure used for the bridge-block problem. The *evert*, *link*, *findpath*, and *condensepath* functions described in the previous section can be modified in a straightforward way to perform the block tree transformations described above. With these functions, the block operations are implemented as follows:

make vertex(A): Return *maketree*(null). (This does not create a block, so the label A is ignored.)

find block(u, v): Return *label*(*findpath*(u, v)).

insert edge(u, v, A): If u and v are in different components, denoted C_u and C_v, respectively, execute $\tilde{r} = maketree(A)$. If $|C_u| \leq |C_v|$, execute *evert*(u), *link*($u, \hat{r}$) and *link*($\hat{r}, v$); otherwise, execute *evert*(v), *link*($v, \hat{r}$) and *link*($\hat{r}, u$). If u and v are in the same component, execute *condensepath*(*findpath*(u, v), A).

The lemmas and analysis given for the bridge-block problem can be easily adapted to the block problem.

THEOREM 2. *Using the above data structure, the on-line block problem can be solved in* $O(n \log n + m)$ *time and* $O(n)$ *space.*

COROLLARY 2. *If the graph is initially connected, the on-line block problem can be solved in* $O(m\alpha(m, n))$ *time.*

4. A Data Structure for Optimal Algorithms. In this section we replace the data structure used in Sections 2 and 3 with a new, more sophisticated data structure called the *link/condense tree*. The link/condense tree is derived from the dynamic tree data structure of Sleator and Tarjan [14], [15]. Using their data structure, a tree T of size n can be everted in time $O(\log n)$, a path of length k between two nodes in T can be found in time $O(k + \log n)$, and the root of T can be linked to a node in another tree of size n' in time $O(\log n + \log n')$. Condensations cannot be performed at all. The goal of the link/condense tree data structure is to speed path-finding, to allow condensation, to support eversion efficiently, and to make the time for a link depend logarithmically on n and not at all on n'. By maintaining the block forest or bridge-block forest with link/condense trees, any sequence of $n - 1$ component links can be performed in $O(n\alpha(m, n))$ time, and it remains the case that $n - 1$ path condensations take time $O(n\alpha(m, n))$. We present two variants of the data structure: one that supports condensation of every node in a path (the bridge-block problem), and one that supports condensation of every other node in a path (the block problem). The variants differ significantly.

We base our data structure on the implementation of dynamic trees described in [15]. The following summary description of dynamic trees is taken from p. 678 of [15] (see Figures 4 and 5).

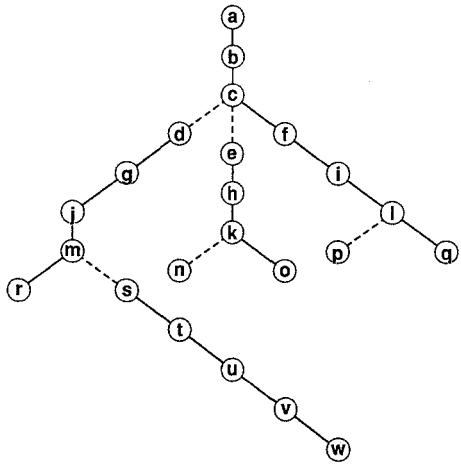

Fig. 4 [15]. A dynamic tree: the actual tree. Dashed edges separate paths corresponding to solid subtrees in the virtual tree.

We represent each dynamic tree T by a *virtual tree* V containing the same nodes as T but having a different structure. Each node of V has a left child and a right child, either or both of which may be a null, and zero or more middle children. We call an edge joining a middle child to its parent *dashed* and all other edges *solid*. Thus the virtual tree consists of a hierarchy of binary trees, which we call *solid subtrees*, interconnected by dashed edges. The relationship between T and V is that the parent in T of a node v is the symmetric-order successor of v in its solid subtree in V, unless v is last in its solid subtree, in which case its parent in T is the parent of the root of its solid subtree in V. In other words, each solid subtree in V corresponds to a

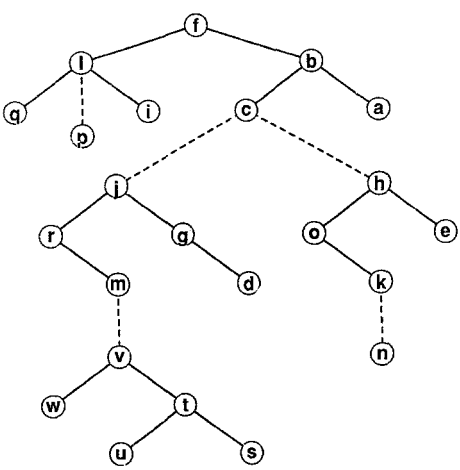

Fig. 5 [15]. The virtual tree representing the actual tree of Figure 4.

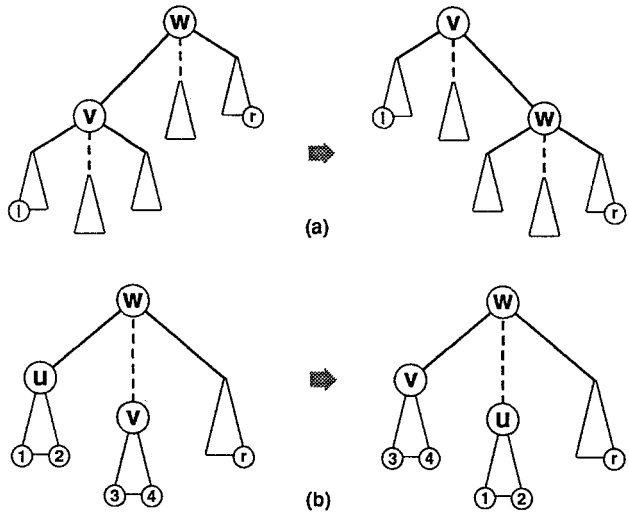

Fig. 6. Rotation and splicing primitives. (a) Rotating the edge between v and w. If w is the root of its solid subtree, then pointers to l and r, the leftmost and rightmost nodes in the subtree, must be added to v. (b) Splicing v to w. Node 3 becomes the leftmost node in the solid subtree rooted at w, while nodes 1 and 2 become leftmost and rightmost, respectively, in the solid subtree rooted at u.

path in T, with symmetric order in the solid subtree corresponding to linear order along the path, from first vertex to last vertex.

The parent of v in T is called the *true parent* of v, and the parent of v in V is called the *virtual parent*. The solid subtrees are maintained with *splay trees* [15]. A *splay at node x* moves x to the root of its solid subtree by applying a standard binary tree rotation to every edge along the path from x to the root (Figure 6(a)). A rotation rearranges left and right children while preserving the symmetric order. Middle children are unaffected. The sequence of rotations is determined by the structure of the path [15]. The splay procedure can be extended to the full virtual tree with the addition of a second primitive called *splicing*, which exchanges a middle child with the left child of a solid subtree root (Figure 6(b)). A *null splice* makes the left subtree of the root into a middle child without replacing it by another left subtree. An *extended splay at node v*, abbreviated *e-splay at v*, moves v to the root of its virtual tree without changing the structure of the actual tree that the virtual tree represents.

The e-splay algorithm is described fully in [15], where it is shown that the amortized cost of an e-splay in a tree of size n is $O(\log n)$. In the dynamic tree data structure, every operation is implemented by $O(1)$ extended splays followed by $O(1)$ additional operations of amortized cost $O(\log n)$. The link/condense data structure differs in that path-finding and condensation do not involve extended splays, while everting and linking do. We augment the dynamic tree data structure to allow path-finding and condensation without significantly interfering with extended splays.

As defined in [15], a dynamic tree node v contains pointers to its left and right children in the solid subtree, *left*(v) and *right*(v), and to its virtual parent, *vparent*(v). It also contains a bit *reverse*(v), used to implement eversion efficiently (see [14]). Node v may enter *reversed state*, in which case the meanings of *left*(v) and *right*(v) are reversed. (That is, *left*(v) points to the right child, and *right*(v) points to the left child.) The reversal state of v is given by the exclusive-OR of the *reverse* values stored in v and all its ancestors in the solid subtree. Rules are given in [15] for updating the reversal bit and left, right, and vparent pointers of the nodes affected by a rotation or splice. Note that the reversal states of a solid subtree root and its children can be determined by examining a constant number of reverse bits. The reversal states of the leftmost and rightmost nodes in a solid subtree can be determined by examining which of their predecessor or successor pointers points back to the root.

In the next two sections we describe the link/condense data structure, first discussing the implementation of eversion and fast linking and then discussing path-finding and condensation. We analyze the running time of the data structure in terms of three primitive operations: rotation, condensation of two nodes, and the finding of the next node along a path. The cost of any sequence of requests can be measured by the number of these primitives performed. After determining the amortized cost of link, evert, findpath, and condensepath in terms of these primitives, we factor in the (nonconstant) running time of the computations associated with each primitive and derive a bound on the overall running time. We prove the following two theorems.

THEOREM 3. *Any sequence of m bridge-connected component operations can be performed in worst-case time $O(m\alpha(m, n))$ and $O(n)$ space.*

THEOREM 4. *Any sequence of m biconnected component operations can be performed in worst-case time $O(m\alpha(m, n))$ and $O(n)$ space.*

5. Eversion and Linking of Link/Condense Trees. An eversion of a link/condense tree T at node u is done using the same procedure as in the dynamic tree data structure (see [14]). Node u is moved to the root of the virtual tree V by an extended splay. Following the e-splay, the right subtree of u contains the path from u to the root of T. A null splice at u is done. The reverse bit in u is toggled, reversing the direction of the entire solid path containing u, and making u the root of the actual tree as well as of the virtual tree. The time for an evert is $O(1)$ plus the time for an extended splay.

A link of T_1 to $v \in T_2$ connects the two trees by making u, the root of T_1, a child of v. The basic dynamic tree procedure is: e-splay u to the root of V_1, e-splay v to the root of V_2, and set *vparent*(u) = v. The time for this procedure is $O(1)$ plus the time for the extended splays in V_1 and V_2, which is $O(\log|V_1| + \log|V_2|)$. We wish the time for a link to be proportional only to the size of V_1, however, so we use the following idea: when V_2 is more than twice the size of V_1 we temporarily defer the extended splay in V_2. Instead, node u is made a middle child of node v by a

special edge called a *deferred link*. Thus T', the tree resulting from linking T_1 and T_2, is represented by two virtual trees V_1 and V_2 connected by a deferred link. The link is completed only when needed to allow a subsequent eversion or link.

The process of adding deferred links may be repeated, so that in general a real tree T is represented by a collection of virtual tree data structures connected by deferred links. For intuition, suppose each virtual tree is shrunk to a single node, and define the weight of such a node to be the size of the corresponding virtual tree. The result is a tree T^d with only deferred links as edges. The weight of a node in T^d is at most half that of its parent. This implies that T^d has depth $O(\log n)$, where n is the size of the corresponding real tree. During an eversion, at most depth of T^d deferred links must be completed. By managing the deferred links carefully, the depth of T^d will remain $O(\log n)$, and the cost of an eversion will not increase significantly.

The notion of the tree T^d is useful for intuition, but it does not fully capture the data structure that will result from including the path-finding and condensation algorithms. For designing these algorithms, it is better to regard the entire data structure representing a real tree T as a single virtual tree V in which some middle edges are marked as deferred links, to be handled specially by the e-splay algorithm. We formalize and generalize the notion of a deferred link in V. The nodes of V are partitioned into a collection D of disjoint *deferred link sets*, which is a coarsening of the partition induced by the solid subtrees. All nodes in a given solid subtree belong to the same deferred link set, but a set may include many solid subtrees. A deferred link is any dashed (middle) edge between two nodes in different sets. The sets are subject to unions and are explicitly maintained with a fast disjoint set union algorithm. We let *dfind*(u) and *dunion*(A, B) denote the standard set union operations applied to D. The disjoint set union algorithm maintains a value denoted *dsize*(S). When *maketree* creates a new single-node tree, a new set S of D is created, containing only the new node, and *dsize*(S) is set to 1. Whenever *dunion*(A, B) occurs, the algorithm sets *dsize*(A) = *dsize*(A) + *dsize*(B).

Let v be a node and let r_v be the parent node of the first deferred link on the path from v to the root of V. If there is no such deferred link, then let r_v be the root of V. A deferred link partition is *valid* if the following properties hold:

I. If two nodes u and v belong to the same set S, then $r_u = r_v$. Node r_v is called the root of S, denoted *D-root*(S). This condition implies that the set $S \cup \{\textit{D-root}(S)\}$ forms a connected subtree.

II. If S_2 is the set containing *D-root*(S_1), then $\textit{dsize}(S_2) \geq 2 \cdot \textit{dsize}(S_1)$, unless *D-root*($S_1$) is the root of V, in which case $S_1 = S_2$.

Our link algorithm will take advantage of the properties of a valid deferred link partition. If no condensations are performed, then each deferred link set S forms a connected subtree by itself, and $|S| \leq \textit{dsize}(S)$. In general, however, it will be the case that S is disconnected and $|S| \leq \textit{dsize}(S)$. Our analysis will be true for any valid partition.

LEMMA 3. *Let node v be an ancestor of node u and let $n = \textit{dsize}(\textit{dfind}(v))$. Then there are $O(\log n)$ deferred links on the path from u to v.*

PROOF. Straightforward application of property II. □

We now give a version of Sleator and Tarjan's extended splay that is modified to handle deferred links. The extended splay is a five-pass process. In the first pass we walk up the path from x to the root of its virtual tree. Each time we encounter a node y that is in a new solid subtree, we splay at y. After the first pass the path from v to the root consists of dashed edges and deferred links. In the second pass we walk up the path, splicing at each dashed edge. After the second pass the path consists of solid subtrees, containing only left children, separated by deferred links. In the third pass we splay at each parent of a deferred link. After the third pass the path from x to the root consists only of deferred links. In the fourth pass we walk up the path, converting each deferred link into a regular dashed edge by uniting the sets that contain the parent and child of the deferred link, and splicing at the resultant dashed edge. After the fourth pass all nodes on the path belong to the same set of the D partition, and x and the root of the virtual tree are in the same solid subtree. In the fifth pass we splay at x, making x the virtual tree root. (See Figure 7.) If the initial path contains no deferred links, then our e-splay reduces to passes one through three; this is exactly the original e-splay of Sleator and Tarjan.

LEMMA 4. *An extended splay at node v maintains a valid deferred link partition.*

PROOF. Assume the partition is valid before the extended splay. Only pass four affects the partition. After pass three each node w on the path from v to the root belongs to a distinct set S_w, whose D-root is either $vparent(w)$ or w itself if w is the virtual tree root. Let S' be the union of all these sets, produced by pass four. The nodes of S' form a connected subtree, and $D\text{-}root(S')$ is the root of the virtual tree. Let S be any set such that $D\text{-}root(S)$ is contained in S' following pass four. This implies that prior to pass four, $D\text{-}root(S)$ was contained in one of the sets that was united to form S'. Since the size of S' is at least the size of any of its constituent sets, property II holds for set S after pass four. □

We now give a precise description of $link(u, v)$, u the root of T_1, $v \in T_2$. First, u is moved to the root of V_1 by an e-splay. We set $vparent(u) = v$, making u a middle child of v by a deferred link. If this link does not violate deferred link property II (i.e., $dsize(dfind(v)) \geq 2 \cdot dsize(dfind(u))$), then we are done. If it does violate the property, then we perform a partial e-splay at v to create a new valid partition. We search up the virtual tree from v until encountering either the root or a deferred link from some node x to node y such that the sum of n and the sizes of the deferred link sets on the path up to x is at most half the size of the set containing y. The subtree rooted at x is temporarily cut out by setting $vparent(x)$ to null. An e-splay at v is done, making v the root of this temporary subtree. We perform $D\text{-}union(D\text{-}find(u), D\text{-}find(v))$, thereby making the deferred link joining u and v into a dashed edge. The subtree is restored to the original tree by setting $vparent(v) = y$, making v a middle child of y by a deferred link. This results in a valid deferred

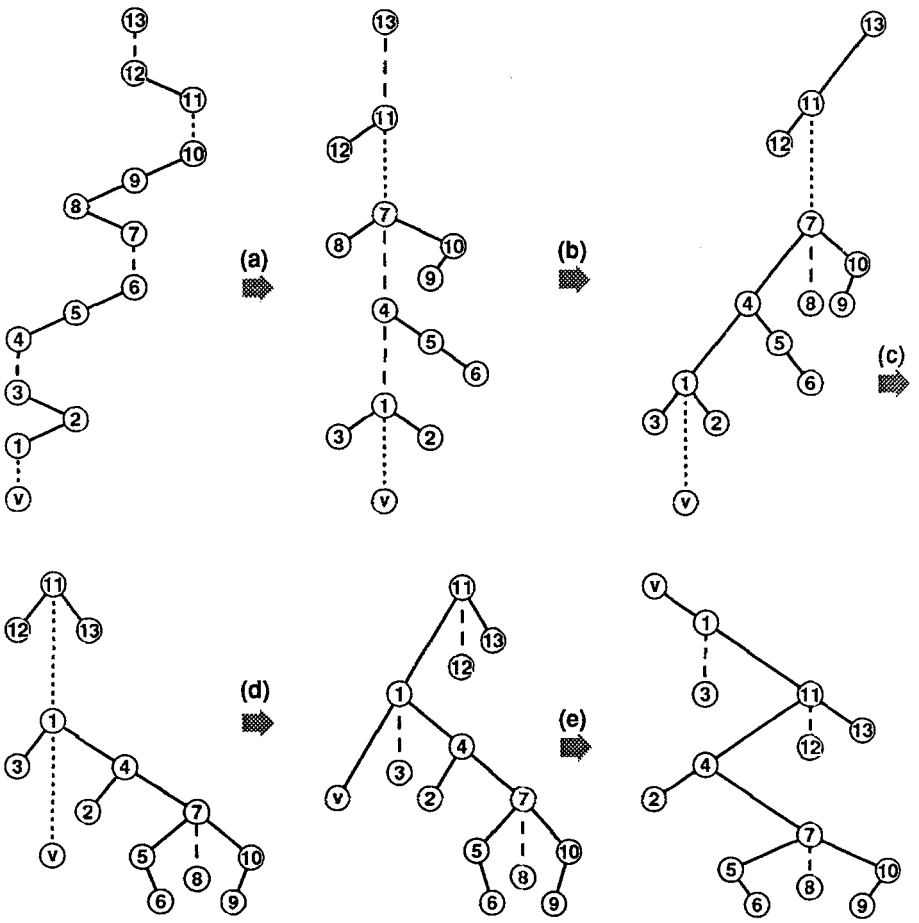

Fig. 7. An extended splay at node v. Subtrees of nodes on the path have been deleted for clarity. Dotted lines represent deferred links. (a) First pass: splaying inside solid subtrees (see [15] for details of splaying). (b) Second pass: splicing dashed edges. (c) Third pass: splaying solid subtrees between deferred links. (d) Fourth pass: converting deferred links to dashed edges, followed by splicing. (e) Fifth pass: splaying along final solid path.

link partition and completes the link. Note that the search for node y can be combined with the first pass of the e-splay at u.

5.1. Amortized Analysis of Evert and Link. To begin the running-time analysis, we observe that the time to perform an eversion or link is $O(1)$ plus the time to do an extended splay. The time for an e-splay is determined by the length of the path traveled up from the starting node. We can measure this length by the number of rotations performed, as stated at the end of Section 4.

To analyze the cost of an operation in terms of the number of rotations, we extend the amortization argument used by Sleator and Tarjan. Let Φ be a *potential function* [15], [22] defined on the virtual tree structure. For each node

x in a virtual tree, we define the weight of node x, $w(x)$, as the number of virtual tree descendants y of x, including x itself, such that $dfind(y) = dfind(x)$, i.e., all descendants of x belonging to the same deferred link set as x. We define the logarithmic weight, $\lg w(x)$, of node x to be $\log w(x)$.[5] Then

$$\Phi = \sum_{\text{nodes } x} 2 \lg w(x) + \sum_{S \in D} a(1 + \log(dsize(S))),$$

where a is a constant to be selected later.

If t is the actual cost of an e-splay, then the amortized cost is defined to be $t + \Delta\Phi$, where $\Delta\Phi$ is the change in the potential function between the start and finish of the operation.

Suppose that at the start of an extended splay at v there are k deferred links on the path from v to the root r of the virtual tree. Then the path passes through $k + 1$ sets $S_0, S_1, \ldots, S_k$ of the D-partition, $v \in S_0$ and $r \in S_k$. Abbreviating $dsize(S_j)$ by n_j, let $p_i = \sum_{0 \le j \le i} n_j$.

LEMMA 5. *For any* $0 \le i \le k$, $p_i \le 2n_i$.

PROOF. From Lemma 3, if $n_i = m$, then $i \le \log m$. By the deferred link invariant, $n_{i-1} \le n_i/2$, $n_{i-2} \le n_{i-1}/2$, etc. Thus

$$p_i = \sum_{0 \le j \le i} n_j \le \sum_{0 \le j \le \log n_i} \frac{n_i}{2^j} \le 2n_i - 1. \qquad \square$$

To bound the cost of an extended splay we consider the passes in groups. After passes one, two, and three, the path from v to the root consists of the $k + 1$ nodes $v = x_0, x_1, \ldots, x_k = r$. These three passes do not change the deferred link partition. Any changes to the potential function involve only the logarithmic weight term. Sleator and Tarjan use this restricted potential function to show that in the basic dynamic tree data structure an e-splay at node v has cost at most $12 \lg' v$, where $\lg' v$ is the logarithmic weight of v after the extended splay [15]. Passes one, two, and three of our modified e-splay essentially treat the sections of path between deferred links as paths in distinct dynamic trees; it is straightforward to adapt the proof of Sleator and Tarjan to show that the cost of these passes is at most

$$\sum_{0 \le i \le k} b \lg' x_i \le \sum_{0 \le i \le k} b \log n_i,$$

where b is a sufficiently large positive constant. Pass four does no rotations so it has zero actual cost, but the unions done in pass four change the potential. The $k + 1$ sets S_i, $0 \le i \le k$, are replaced by a single set S' with size p_k. The unions also increase the weight of each node x_i on the path, since the number of descendants of x_i that belong to the same set as x_i increases. Only the nodes on

[5] By log we mean continuous binary logarithm.

the path increase in weight as a result of the unions, since the nodes of set S_i together with $x_{i+1} = D\text{-}root(S_i)$ form a connected subtree. Node x_i increases in weight by at most p_i. Thus the cost of pass four is at most

$$a \log p_k + \sum_{0 \le i \le k} 2 \log(p_i) - a(1 + \log n_i).$$

Pass five has cost at most $b \log p_k$, again by applying the proof by Sleator and Tarjan.

By Lemma 5, $\log p_i$ is at most $1 + \log n_i$. Therefore we can choose a sufficiently large that the terms in the sum are nonpositive, and the total cost of the e-splay is $(a + b) \log p_k$.

LEMMA 6. *Let u be contained in a virtual tree V of n nodes. The amortized cost of evert(u) is $O(\log n)$. If u is the root of V, the amortized cost of link(u, v) is also $O(\log n)$.*

PROOF. An extended splay at u has amortized cost $O(\log n)$, since $p_k \le n$. This implies the bound on the cost of an eversion.

The total cost of *link*(u, v) consists of the cost of the e-splay at u, which is $O(\log n)$, and possibly the cost of a partial e-splay at v and a unification of the deferred link sets containing u and v. Since the unification increases the weight of v by at most n, it also has amortized cost $O(\log n)$. To prove the lemma, we must show that the partial e-splay at v has cost $O(\log n)$.

The extent of the partial e-splay is determined by walking up from v until finding a deferred link set S_k such that $2(n + p_k) \le n_{k+1}$. This implies that, for each set affected by the e-splay at v,

$$n_i < 2(n + p_{i-1}). \tag{1}$$

Let ℓ be minimal such that $n_\ell > n$. If there is no such ℓ, then $2n \ge 2n_k > p_k$ (by Lemma 5) and the cost of the e-splay, and hence the link, is $O(\log n)$.

If ℓ is defined, we write the total cost of the e-splay as $A + B$, where

$$A = b \log n_\ell + \sum_{0 \le i < \ell} b \log n_i + 2 \log p_i - a(1 + \log n_i)$$

and

$$B = (a + b) \log p_k - a \log n_k + \sum_{\ell < i \le k} b \log n_i + 2 \log p_i - a(1 + \log n_{i-1}).$$

Sum A is $O(\log n_\ell)$, since a can be chosen sufficiently large that the terms under the sum are nonpositive. By Lemma 5, $p_{\ell-1} \le 2n_{\ell-1}$, and by the definition of ℓ, $n_{\ell-1} < n$. Combining these observations with (1) we have

$$n_\ell < 2(n + p_{\ell-1}) < 2(n + 2n_{\ell-1}) < 6n.$$

Hence A is $O(\log n)$.

Using Lemma 5 we can write

$$B \leq O(1) + \sum_{\ell < i \leq k} c \log n_i - a(1 + \log n_{i-1}),$$

where c is a constant that depends on b but not a. By the definition of ℓ and by the deferred link invariant $n_i \geq 2n$ for $i > l$. Using (1) and Lemma 5, this implies $n_i < 2(n + p_{i-1}) \leq 5n_{i-1}$. Hence

$$B \leq O(1) + \sum_{\ell < i \leq k} c \log(5n_{i-1}) - a(1 + \log n_{i-1}).$$

Again it is possible to choose a sufficiently large that the terms under the sum are nonpositive. Therefore, the extended splay at v has amortized cost at most $O(\log n)$. □

LEMMA 7. *Using the fast-linking data structure described above, $n - 1$ component links that combine an initial collection of n single-node components into one final component require $O(n)$ rotations.*

PROOF. By Lemma 6 the amortized cost of a component link is $O(\log n)$ rotations, where n is the number of vertices in the smaller component. This gives the following recurrence for the total amortized cost of component links:

$$A(n) \leq \max_{1 \leq j \leq n/2} \{A(j) + A(n - j) + c(1 + \log j)\}.$$

This implies $A(n) = O(n)$. The initial value of Φ is zero, and Φ is always positive. By a standard property of amortized analysis [22], this implies that $A(n)$ bounds the actual number of rotations. □

Note that the time per rotation is $O(1)$ plus the time for $O(1)$ changes to pointers plus the time for $O(1)$ operations on the deferred link sets.

6. Path-Finding and Condensation in Link/Condense Trees. We begin with the implementation of *findpath*(u, v). To each tree node v we add a pair of pointers *pred*(v) and *succ*(v) that point to the symmetric-order predecessor and successor, respectively, of v in its solid subtree. In general, *succ*(u) points to the true parent of u. We also add a pair of pointers *leftmost*(v) and *rightmost*(v). These contain null values except when v is the root of a solid subtree, at which time they point to the leftmost node l and rightmost node r in this solid subtree. The pointers *pred*(l) and *succ*(r) point to the root of the solid subtree, so that we may move between leftmost, root, and rightmost nodes in $O(1)$ pointer steps. The meanings of *leftmost*(v) and *rightmost*(v), and *pred*(v) and *succ*(v), are switched if v is in reversed state.

Since rotation preserves symmetric order, it does not affect the values of *pred*(v) or *succ*(v). If the rotation replaces the old subtree root r with a new root v, however,

then the values of *leftmost*(*r*) and *rightmost*(*r*) must be copied to *v*, and *pred*(*leftmost*(*r*)) and *succ*(*rightmost*(*r*)) must be updated to point to *v*. A splice causes more significant changes to the new pointers. Let *w* be the root of a solid subtree, let *u* be the (possibly null) left child of *w*, and let *v* be a middle child of *w*. After splicing *v* into *w*, the new pointers are updated as follows (a prime indicates the new value):

$$\begin{aligned} leftmost'(u) &= leftmost(w), & pred'(leftmost'(w)) &= w, \\ leftmost'(w) &= leftmost(v), & pred'(leftmost'(u)) &= u, \\ rightmost'(u) &= pred(w), & succ'(pred'(w)) &= w, \\ pred'(w) &= rightmost(v), & succ'(rightmost'(u)) &= u. \end{aligned}$$

In the above, by *pred*(*v*), *succ*(*v*), *leftmost*(*v*), and *rightmost*(*v*) we mean the actual predecessor, successor, etc., of *v*. The fields containing these values may be switched if node *v* is in reversed state.

Recall that *findpath*(*u*, *v*) returns a path *P* that is a list of pointers to nodes in the order they occur along the tree path in *T* from *u* to *v*. As in Sections 2 and 3, this path is found by generating the paths from *u* and from *v* to the root of *T* in lock-step, one node at a time, stopping as soon as one path intersects the other at the nearest common ancestor of *u* and *v*. Thus the path-finding problem reduces to the problem of generating the path from node *v* to the root of *T*, one node a time.

First we consider the problem of finding the path from *v* to the farthest ancestor *r* of *v* in the same solid path of *T*. In the corresponding solid subtree of *V*, *r* is the rightmost node. The path from *v* to *r* can be found by traversing the solid subtree in symmetric order from *v* to *r*. This can be done in $O(|P|)$ pointer steps by following successor pointers from *v*. Unfortunately, it is impossible to determine the reversal state of *v*, and hence the interpretation of *pred*(*v*) and *succ*(*v*), without walking down to *v* from the solid subtree root. This would increase the cost of path finding to $O(d + |P|)$, where *d* is the depth of *v*. It is possible, however, to generate a path that consistently moves in the same direction once a direction in which to start is chosen. If *succ*(*x*) points to node *y*, then one of *pred*(*y*) or *succ*(*y*) must point back to *x*. The other pointer of *y* points to the next node in the path. Using this technique, a path from *v* can be generated that follows either only successor or only predecessor pointers. Eventually this path will encounter the leftmost or rightmost node of the solid subtree, followed immediately by the root node. Since the reversal state of the root is known, the direction of the path can be determined.

To find the path from *v* to *r*, two tentative paths out of *v* are generated in lock-step, one starting at *succ*(*v*) and one starting at *pred*(*v*). When one path encounters a solid subtree root, the true direction of both paths is determined. The path of predecessors is discarded and the path of successors is continued, if necessary, until *r* is reached. At most two nodes are examined for each node of the final path, so the total number of pointer steps is $O(|P|)$.

Suppose the path *P* from *v* to the root of *T* passes through *k* solid paths. For the purposes of path-finding, it is unnecessary to distinguish between deferred links and regular dashed edges from middle children. Let r_i denote the highest node in

the ith solid path, $1 \leq i \leq k$, or equivalently, the rightmost node in the corresponding subtree of V. Thus r_k is the root of T. The true parent of r_i is some node v_{i+1} in the $(i+1)$st solid path. A path P^i from $v = v_1$ to r_i can be extended to a path P^{i+1} from v to r_{i+1} by concatenating P with the path from v_{i+1} to r_{i+1}. The true parent of r_i is given by $vparent(succ(r_i))$, and so can be found in $O(1)$ pointer steps. This implies that the complete path $P = P^k$ can be found, one node at a time, in $O(|P|)$ pointer steps.

To analyze the cost of the *findpath*(u, v) algorithm we define each step to be a unit-cost primitive, equivalent to a rotation.

LEMMA 8. *The path P between two nodes u and v can be found with amortized cost $O(|P|)$.*

PROOF. Since the data structure is not modified, there is no change in Φ. The *Findpath* algorithm generates in lock-step the paths P_1 and P_2 from u and v, respectively, to the root of T. At each step, there is a known section of P_1 from u to some node x and two tentative paths being generated out of x. The same holds for P_2 and some known node y. Path finding terminates when some tentative path X intersects a known section. Then P consists of the two known sections plus X. Since each step generates at least one node on P, there are at most $|P|$ steps. Hence the amortized cost (and worst-case cost) is $O(|P|)$. □

COROLLARY 3. *A* find block *operation has amortized cost and worst-case cost $O(1)$.*

6.1. Condensation in Link/Condense Trees. There are two variants of *condensepath*(P, A) to consider: one that condenses every node on the path into a single node, suitable for the bridge-block problem, and another that condenses every other node on the path, suitable for the block problem. In both cases, we use the condensible node data structure to implement the tree nodes. For the purposes of condensation, it is unnecessary to distinguish between deferred links and regular dashed edges from middle children. In general a condensation increases the number of deferred links, but we will show that the condensation algorithms maintain a valid deferred link partition.

We begin with the first variant. Let *condense2*(x, y, A) be a function that condenses a path consisting of two adjacent nodes x and y, and labels the resultant node z by A. *condensepath*(P, A) proceeds by repeating the following *condensation step* until P consists of a single node: select a pair of adjacent nodes x, y from P. Perform *condense2*(x, y, A). (Let T' be the tree resulting from this condensation.) Replace x and y in P with z.

Since path condensation preserves adjacency, it is clear that after a condensation step, P is the path between u and v in tree T'. An inductive proof using this observation shows that *condensepath* correctly condenses a path of arbitrary length.

In implementing *condense2*(x, y, A) care must be taken to avoid increasing the number of descendants of any node. Such an increase would raise the value of the

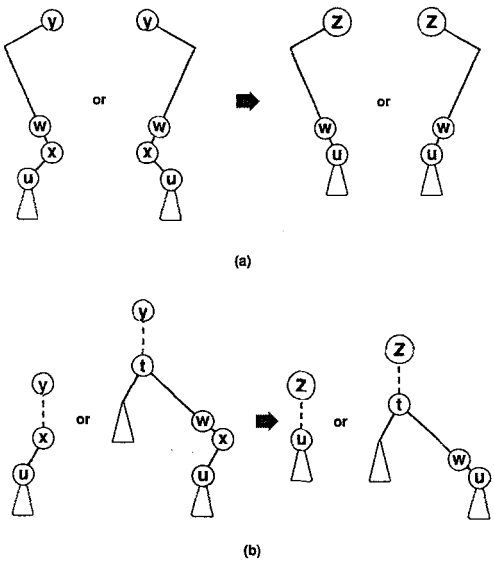

Fig. 8. Cases of *Condense2*(x, y, A). (a) Case 1. $w = vparent(x)$. (We may have $w = y$.) Node z is the result of the condensation. (b) Case 2. (We may have $t = w$.)

potential function and add to the amortized cost. We distinguish two cases based on the relationship between x and y in the virtual tree:

1. x and y are in the same solid subtree.
2. x and y are in different solid subtrees.

In case 1 at least one of $pred(x)$, $succ(x)$ contains a pointer to y. In case 2 neither of these fields contain a pointer to y. This fact can be used to determine which case applies.

Case 1 is shown in Figure 8(a). The adjacency of x and y in the actual tree implies that one is the ancestor of the other in the solid subtree. Assume that y is the ancestor of x. For the moment, we assume that the reversal states of x and y are known. Let y be the successor of x. (The other possibility is symmetric.) This implies that the right subtree of x is empty, and that the successor pointer of x points to y. By examining the fields of x and y for this configuration, we can determine which of them is in fact the ancestor. That is, y is the ancestor of x if $right(x)$ is null and $succ(x)$ is y, or, in the symmetric case, $left(x)$ is null and $pred(x)$ is y.

Let $w = vparent(x)$. (It may be the case that $w = y$.) Let $u = left(x)$. To begin the condensation, node u is made a child of w in place of node x. This is done by replacing the pointer to x in w by a pointer to u and setting $vparent(u) = w$. The old value of $reverse(u)$ is replaced by the exclusive-or of $reverse(u)$ and $reverse(x)$. Nodes x and y are combined to form z. The fields in $N(z)$ are copied from $N(y)$, with the exception of $label(z)$, which is set to A, and of $pred(z)$, which is copied from $N(x)$.

Combining x and y, which unites their subnode sets, guarantees that all former pointers to x or y are now pointers to z. The former predecessor of x and the former successor of y become the predecessor and successor, respectively, of z. All former middle children of x or y become middle children of z. Hence, node adjacencies in the transformed tree are correctly preserved. In general, the reversal states of x and y are not known, but they are not actually needed. One of $left(x)$ or $right(x)$ must be null, so the pointer to u can be found in the other. Similarly, one of $pred(y)$ or $succ(y)$ points to x. The corresponding field in z is copied from whichever of $pred(x)$ or $succ(x)$ does not contain a pointer to y.

Case 2 is shown in Figure 8(b). The adjacency of nodes x and y implies that one, say x, must be rightmost in its solid subtree. The root of this solid subtree is a middle child of y. The reversal state of x can be computed by determining which of $pred(x)$ or $succ(x)$ points to the root of the solid subtree. Since x is rightmost, its right subtree is empty. Let $u = left(x)$. If x is the root of its subtree, a null splice is performed to make u a middle child of x. Otherwise, let $w = vparent(x)$. Using the pointer and reversal bit updates given in case 1, u replaces x as the right child of w. This makes $pred(x)$ rightmost in the solid subtree. The leftmost field in the solid subtree root and the successor field in $pred(x)$ (i.e., the field containing a pointer to x) must be updated accordingly. Finally, x and y are condensed to give z. $N(y)$ is copied to $N(z)$, and z is labeled A. Note that this makes z a member of the deferred link set that initially contained y.

LEMMA 9. *Condense2*(x, y, A) *preserves a valid deferred link partition.*

PROOF. Assume that the partition is valid prior to applying *condense*2. The condensation can only affect the sets which contain x and y and the sets for which x or y is the set root.

Suppose x and y initially belong to the same deferred link set S. This is always true in case 1 of *condense*2(x, y, A), and may be true in case 2. Then S remains a valid deferred link set, since it still forms a connected subtree with $D\text{-}root(S)$ and the value of $dsize(S)$ is unchanged (alghough $|S|$ decreases by one).

In case 2, however, x and y may belong to two different sets, say S_1 and S_2, respectively. This situation occurs when the root of the solid subtree containing x is a child of y by a deferred link. This implies that y is $D\text{-}root(S_1)$, that y and S_1 form a connected subtree, and that $dsize(S_2) \geq 2dsize(S_1)$. We claim that S_1 remains a valid set with new root z. All middle children of x that belong to S_1 become middle children of z, as do all middle children of y that belong to S_1. The left subtree of x remains in the solid subtree that contained x. Therefore z and S_1 form a connected subtree with $z = D\text{-}root(S_1)$. Since $z \in S_2$, and no dsize value is changed, property II is satisfied for S_1.

Suppose that prior to the condensation x (or y) is $D\text{-}root(S')$ for some set S' that contains neither x nor y. Then x and S' form a connected subtree. The condensation makes all middle children of x (and y) into middle children of z, so property I is preserved for S', with $D\text{-}root(S') = z$. The dsize of the set containing z is at least the $dsize$ of the set initially containing x (or y), so property II must hold for S'. □

We define the *condense*2 algorithm to be a unit-cost primitive, like rotation.

LEMMA 10. *The* condense2 *algorithm has amortized cost* 1.

PROOF. Since the actual cost is 1, we need only bound the change in potential. Since y is originally an ancestor of x, the combining of x and y cannot increase the weight of any tree node. Therefore Φ is nonincreasing, and the amortized cost is at most 1. □

We can now prove Theorem 3. The total cost of any sequence of operations can be measured by the number of rotations, *condense*2 operations, and path-finding steps done. Hence each of these is defined to be a unit cost primitive. The amortized cost of an operation is the number of primitives done plus the change in potential.

The amortized cost of n *maketree* operations is $O(n)$, since each *maketree* creates a new node and a new deferred link set, increasing the potential by a constant amount. By Lemma 7 the total cost of links and eversions over $n - 1$ component links is $O(n)$. By Lemmas 8 and 10 the total cost of path-finding and condensations over $n - 1$ condensing edge insertions is $O(n)$. By Corollary 3 each *find block* has cost $O(1)$. Collecting terms, the total cost of m operations is $O(m)$.

Each primitive operation performs $O(1)$. Collecting terms, the total cost of m operations is $O(m)$.

Each primitive operation performs $O(1)$ condensible node operations. In addition, each rotation involves $O(1)$ operations on the deferred link sets. Thus the worst-case running time for a sequence of m operations is $O(n\alpha(m, n))$ plus $O(n\alpha(O(n), n))$, which is $O(m\alpha(m, n))$.

All data structures are size $O(n)$, and thus the space required by the algorithm is $O(n)$.

6.2. Condensation of Alternate Nodes. The variant of the link/condense tree data structure used for the block problem is complicated by the problem of condensing only the round nodes in a path of alternating round and square nodes. Care must be taken in restructuring the virtual tree so that movement of the square nodes does not increase the value of the potential function, increasing the amortized time per update.

To implement *condensepath*(P, A), we use the function *condense*3(r, u, s, A), which transforms a three-node path consisting of round node r, square node u, and round node s, and returns the round node t, labeled A, resulting from the condensation of r and s. Path condensation proceeds by repeatedly selecting a three-node path from P and replacing it with the single node returned by *Condense*3. The process terminates when P consists of a single round node.

In the virtual tree, one of the three nodes r, u, and s must be the nearest common ancestor of the other two. A problem arises if the square node u is the nearest common ancestor. In this situation, we know of no way to combine r and s without increasing the value of Φ by $O(\log n)$ in the worst case. Therefore, we impose *solid subtree restrictions* on the data structure to ensure that the nearest common

ancestor is never the square node. The splay and splice routines must be modified to guarantee that the restrictions are not violated during an extended splay. These modifications will be discussed in a later section. The solid subtree restrictions are:

(i) *Each solid path in the actual tree either is a single round node or terminates in a square node (i.e., a square node is rightmost in the corresponding solid subtree.)*
(ii) *All square nodes are leaf nodes of their solid subtrees.*

There are four cases of *condense*3(r, u, s, A):

1. All three nodes are in the same solid subtree.
2. u and one round node are in one subtree whose root is a middle child of the other round node.
3. u and one round node are in one subtree, and the other round node is a middle child of u.
4. All three nodes are in different solid subtrees.

Which case applies can be determined by examining *vparent* pointers.

In cases 1 and 2 the relationship between virtual tree and actual tree implies that, in the actual block tree, one round node is an ancestor of u and the other is a descendant of u. In case 1 restriction (i) implies that in the solid subtree one of r and s is a descendant of the other, and u is a child of the descendant round node. This fact can be used to determine which of r and s is the ancestor. Let r be the descendant. (The subcase with r the ancestor is analogous.) Node u is either the left or right child of r. The other child of r is made a child of *vparent*(r) using the pointer and reverse bit updates described in case 1 of the *condense*2 routine of Subsection 6.1. Then nodes r and s are condensed together to give node t, with the fields of t being copied from the fields of s. This makes u a middle child of the condensed node.

In case 2 assume that r is in the same solid subtree as u. (The case of s in the same solid subtree is analogous.) Restriction (i) implies that u is the right child of r and that u is rightmost in its solid subtree. As in case 2 of *Condense*2, the left child of r is made into the right child of *vparent*(r). Then nodes r and s are condensed together to give node t, with the fields of t being copied from the fields of s. This makes u a middle child of the condensed node.

Cases 3 and 4 are simpler. At least one of the round nodes r and s is a middle child of u. By restriction (ii) this round node must be a singleton solid subtree. The two round nodes are simply condensed to give t. The fields of t are copied from whichever round node is not a descendant of u, or, if both r and s are descendants, from either one.

The four cases are shown in Figure 9(a)–(d). From examination of the cases, it is clear that *condense*3 does not violate the solid subtree restrictions.

The proof of Lemma 9 can be adapted to show that *condense*3 maintains a valid deferred link partition. If either r or s is a set root before the condensation, it remains a valid root for the same set after the condensation. Similarly, the proof of Lemma 10 can be adapted to show that *condense*3 has amortized cost $O(1)$.

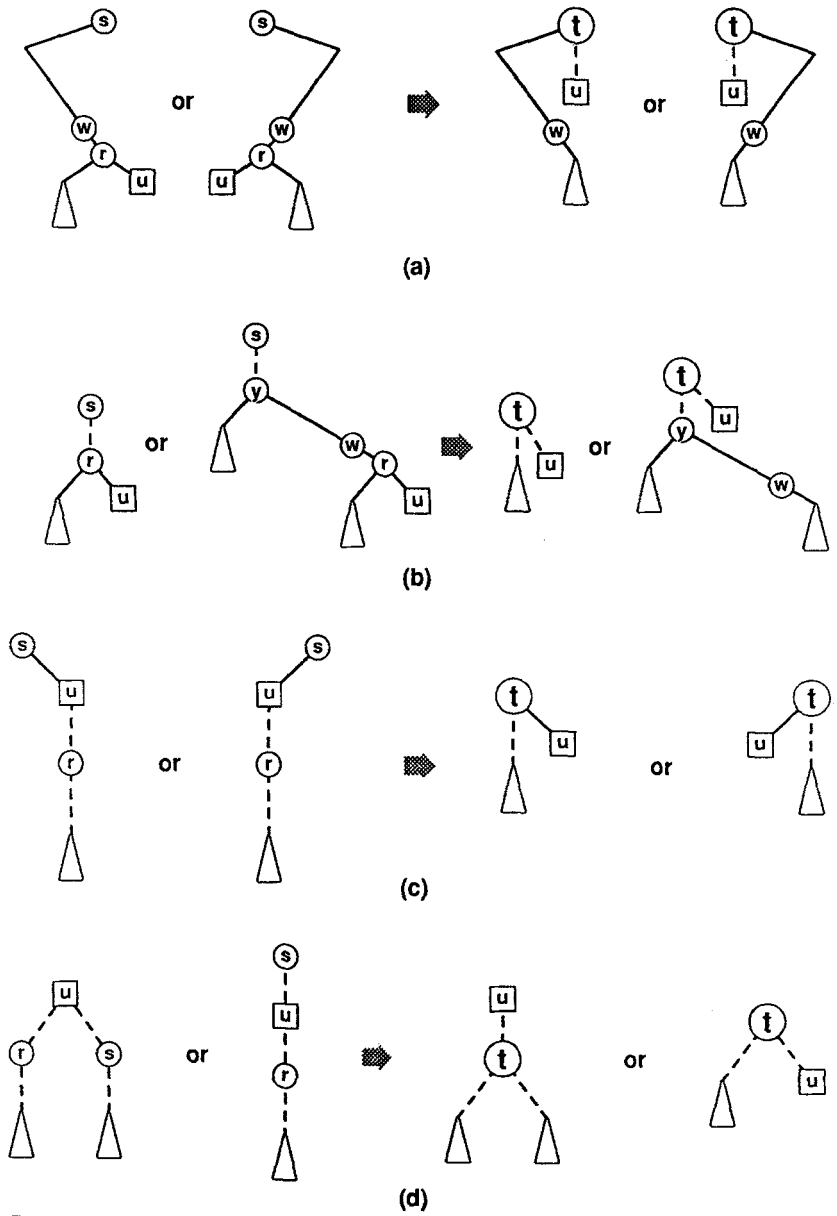

Fig. 9. Cases 1–4 of *Condense3*(r, u, s, A), condensing r and s to give t. Triangles denote subtrees with square rightmost nodes.

6.3. Combining Linking with Alternate Condensation. To produce a link/condense data structure for the block problem, the extended splay procedure that forms the basis for eversion and linking must be changed to support the solid subtree restrictions required for alternate condensation. We show this can be done without increasing the cost of eversion or linking.

Recall that the extended splay is based on *splaying*. A splay at node x moves x to the root of its solid subtree by rotating every edge along the path from x to the root. Splaying does not rearrange the subtrees rooted at nodes off the path from x to the root. This implies that if x is round, a splay at x cannot violate restrictions (i) or (ii). Splaying a square node to the root, however, will violate restriction (ii) except in the trivial case that the square node is a singleton solid subtree. Therefore, we define the function *square-splay*(v), which moves a square node to within one step of the root:

```
square-splay(v) begin
  splay at pred(v);
  splay at succ(v);
  rotate the edge between v and pred(v);
  perform a null splice at pred(v);
  perform a null splice at v;
end
```

In the above, by *pred*(v) and *succ*(v) we mean the actual predecessor and successor, respectively, of v. The fields containing these values may be switched if node v is reversed. (The reversal states of all nodes along the splay path are computed by an initial walk down the path.) The behavior of splaying is such that after the first two splays, v is a right child of *pred*(v) and *pred*(v) is a left child of *succ*(v) (see [15]). If v has no predecessor or successor, then the code involving the missing node can simply be ignored. For example, if v has no predecessor, then we need only splay at the successor. At the conclusion of *square-splay*(v), node v remains a leaf; either it is a singleton solid subtree or it is the left child of the root. Thus *square-splay*(v) does not violate the solid subtree restrictions.

We must also be careful when splicing. An attempt to splice a round child to a square parent may violate restriction (ii), while an attempt to splice any child into a round node that forms a single-node subtree may violate restriction (i). To handle the first case, we introduce a new splice function, *square-splice*(r), where r is a round child of a square parent. Note that by restriction (i), r is a single-node solid subtree.

```
square-splice(r) begin
  let v = vparent(r);
  splice r to v;
  rotate the edge between r and v;
end
```

At the conclusion of *square-splice*(r), square node v is the right child of r. Since r has no right child initially, v remains a leaf node after the rotation. Thus *square-splice* does not violate the solid subtree restrictions.

In the second case we can allow the splicing of a left subtree to a round node r that forms a single-node subtree as long as r has a parent. Since this parent must be a square node, r will immediately participate in a square-splice that will give

it a square right child, guaranteeing that restriction (i) is not violated. The splice is disallowed if r is the root of the virtual tree (and hence the root of the actual tree), or if there is a deferred link from r to its parent.

As before, an extended splay at v is a five-pass process. (In the block algorithm we only e-splay at square nodes.) We use square-splays and square-splices as needed, and do not splice at singleton round nodes from which there are deferred links. After the first pass the path from v to the root consists of dashed edges, deferred links, and solid edges between a square node and its parent. After pass three the path from v to the root consists primarily of deferred links. Between each pair of deferred links there is a section of path that consists of either a single round node, or a solid edge between a square node and its round parent, or a solid edge between a square node and its round parent followed by a dashed edge leading into a single-round-node round subtree. After passes four and five v is either at the root, is a left or middle child of the root, or is the left child of a middle child of the root. Figure 10 gives an example of the first three passes of an extended splay, showing square-splays and square-splices.

A tree can be everted at a round node r by the same procedure used before. To evert at a square node v, we perform an e-splay at v and apply one of the following cases. If v is the root of its virtual tree, no processing is needed to do the eversion. If v is the left child of the virtual tree root, then we simply toggle the reverse bit in the root. If v is a middle child of the root, we perform an ordinary splice at v and toggle the reverse bit in the root. If v is a left child and $vparent(v)$ is a middle child of the root, then we perform an ordinary splice to make $vparent(v)$ a left child of the root, and toggle the reversal bit in the root. In the latter three cases v becomes the end of a solid subpath, so restriction (i) is not violated.

The algorithm for $link(u, v)$ is essentially the same as that of Section 5. The modifications to splaying and splicing do not affect the way the deferred link partition of the virtual tree is maintained and we can show that e-splay and link maintain a valid deferred link partition. It remains to bound the cost of evert and link.

As in Section 5, suppose that at the start of an extended splay at v there are k deferred links on the path from v to the root r of the virtual tree. The path passes through $k+1$ deferred link sets $S_0, S_1, \ldots, S_k$. Let $n_j = dsize(S_j)$ and let $p_i = \sum_{0 \le j \le i} n_j$. We will show that passes one, two, three, and five of the e-splay have cost at most

$$b_1 \log p_k + \sum_{0 \le i \le k} b_1 \log n_i$$

and that pass four has cost at most

$$a \log p_k + \sum_{0 \le i \le k} b_2 \log(p_i) - a(1 + \log n_i)$$

for some constants b_1 and b_2 that are independent of a. The rest of the results of Section 5 follow from this, and we can conclude that $n-1$ component links have

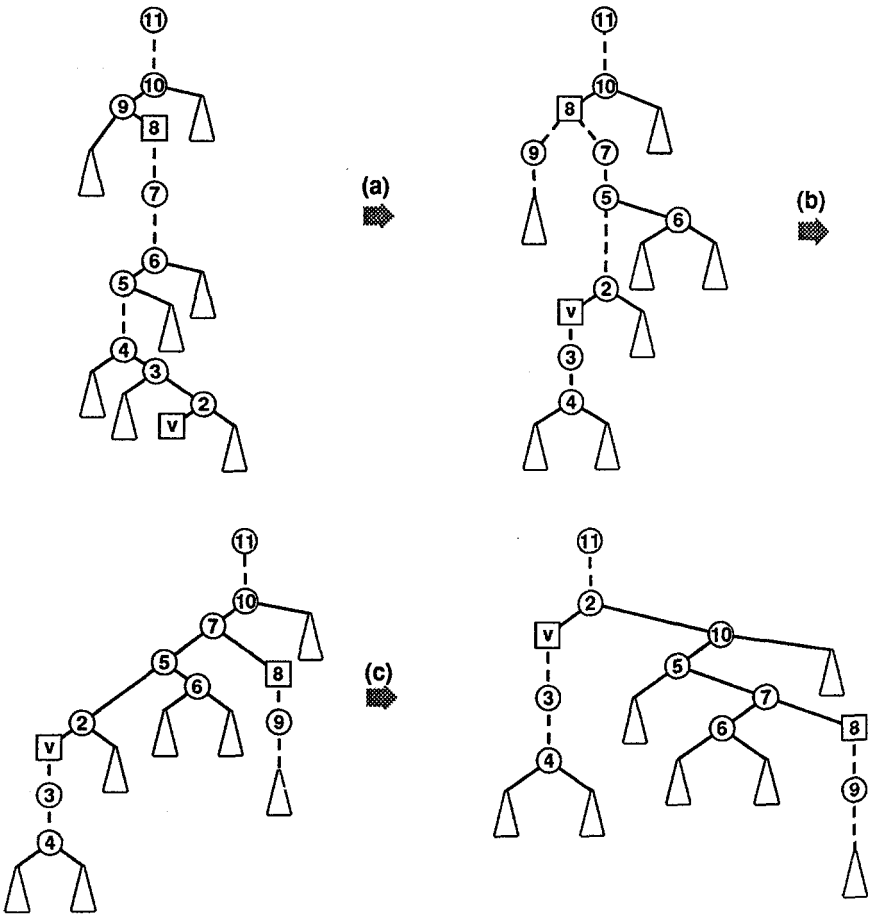

Fig. 10. Passes 1–3 of an extended splay at node v. The path contains no deferred links. Triangles represent subtrees with square rightmost nodes. (a) First pass: splaying and square-splaying inside solid subtrees. (b) Second pass: splicing and square-splicing dashed edges. (c) Third pass: square-splay along final solid path.

cost $O(n)$. Our analysis will be fairly terse, relying on results proven in [15]. The reader is advised to examine that paper for further details and explanation.

We define

$$\Phi = \sum_{\text{nodes } x} 5 \lg w\, x + \sum_{S \in D} a(1 + \log dsize(S)).$$

From Lemma 1 of [15] we conclude the following: let x be a left or right child of y. A single rotation of the edge between x and y has amortized cost $1 + 3(5 \lg w(y) - 5 \lg w(x))$. This bounds the cost of a square-splice.

From the same source, we conclude a second fact: let y be the root of a solid subtree that contains node x. The amortized cost of a splay at x is at most $1 + 3(5 \lg w(y) - 5 \lg w(x))$. In executing *square-splice*(v) we perform two splays and

a single rotation. The properties of symmetric order imply that, since v is a leaf, both the predecessor and the successor of v must at all times be ancestors of v, and hence their logarithmic sizes are always greater than that of v. This implies that the amortized cost of *square-splay*(v) in a solid subtree rooted at t is at most $3 + 45(\text{lgw}(t) - \text{lgw}(v))$.

We now bound the cost of the first three passes of an extended splay at node v.

LEMMA 11. *Consider a section of the path from v to the root that is bounded by deferred links, i.e., all nodes on the path are in the same deferred link set. Let x be the lowest node on the section of path and let t be the highest node. (Thus x is a parent of a deferred link and t is a middle child of a deferred link.) The cost of passes one, two, and three in this section of path is at most* $b_1 \text{ lgw } t$ *for some constant* b_1.

PROOF. Let k be the number of solid subtrees on the initial path from x to t. The amortized cost of the first pass is at most $3k + 45 \text{ lgw } t$. This bound follows from summing the splay or square-splay costs within each solid subtree. The amortized cost of the second pass is at most $k + 15 \text{ lgw } t$, since there are most k square-splices and the sum again telescopes. Thus the amortized cost of first two passes is $4k + 60 \text{ lgw } t$.

If x is square, then, after the second pass, x is at depth k, and in the third pass, *square-splay*(x) will perform exactly one splay, moving the parent (successor) of x to the root (or to within one step from the root, if t forms a round single-node subtree). If x is round, x is at depth $k - 1$ and is itself splayed to the root in the third pass. Therefore, at least $k - 2$ rotations occur in the third pass. We charge 5 for each of these $k - 2$ rotations; the additional charge accounts for $4k - 8$ of the rotations left over from the first two passes. From the discussion of e-splay in [15], we find that even with this additional charge, the amortized cost of the final splay is at most $5 + 15 \text{ lgw}(t)$. Summing over the three passes, we obtain the desired result. □

The increase in the lgw term of Φ during pass four is greater here than in Section 5, because between each deferred link there may be as many as three nodes each of which has its weight increased. Nevertheless, the increase Φ due to the nodes in the ith set on the path is at most $3(5 \log p_i)$. Therefore the cost of pass four is bounded as desired.

From this point the proof of Theorem 4 follows the proof of Theorem 3.

7. General Lower Bounds. Lower bounds for the on-line block and bridge-block problems can be obtained using simple reductions from the disjoint set union problem. Let n be the number of elements and let m be the number of operations in an instance of disjoint set union. Tarjan [20] gave a lower bound of $\Omega(\alpha(m, n))$ on the amortized time per operation and Blum [4] gave a lower bound of $\Omega(\log n/\log\log n)$ on the worst-case time of a single operation. Both these lower bounds apply to the class of separable pointer algorithms for set union [4], [20]. La Poutré [11] has proven the $\Omega(\alpha(m, n))$ amortized cost bound in a general pointer

machine model, and Fredman and Saks [8] have shown an $\Omega(\alpha(m, n))$ bound on the amortized cost per operation and an $\Omega(\log n/\log\log n)$ bound on the worst-case cost per operation in the cell-probe model of Yao [26]. In this powerful and general model, memory is organized into cells, each of which can hold $\log n$ bits. In answering a query, a cell-probe algorithm is allowed to access cells randomly based on the information gathered from previous probes. The pointer-machine and cell-probe models are related as follows: if the number of bits in a separable pointer-machine node is bounded by $\beta(n) \geq \log n$, then the separable pointer machine can be simulated by a cell-probe machine, with running time increasing by a factor of $\beta(n)/\log n$.

The reduction from disjoint set union to the on-line block and bridge-block problems is straightforward. For each set element a we create a pair of vertices a_0 and a_1 connected by an edge e_a. To answer a find query we execute *find block*(a_0) or *find block*(a_0, a_1), depending on whether the reduction is to the bridge-block or block problem, respectively. To unite the sets containing elements a and b we add an edge between a_0 and b_0 and an edge between a_1 and b_1. Thus each set corresponds to a component that is both bridge-connected and biconnected. We define a separable pointer algorithm for bridge-connectivity or biconnectivity to be an algorithm that uses a linked data structure in which no pointer connects the subgraphs representing each graph component. If we begin with a separable algorithm for bridge-connectivity or biconnectivity, the above reduction gives a separable pointer algorithm for disjoint set union. Similarly, the reduction can be used to give a cell-probe algorithm for disjoint set union.

We can also give a reduction of disjoint set union to the variant of the block problem in which the graph is initially connected. The initial graph resembles a wheel with hub vertex h. For each element a there is a vertex v_a and an edge connecting v_a to h. Queries are answered by executing *find block*(h, v_a). To unite the sets containing a and b, an edge is added between v_a and v_b. For the variant of the bridge-block problem in which G is initially connected, we know only the trivial lower bound of $\Omega(1)$ on the time per operation.

8. Remarks and Recent Related Work. One major difference between the dynamic tree data structure of Sleator and Tarjan and the link/condense tree data structure presented here is that in the latter the time to link together two trees is dependent only on the size of the child tree. If condensation is not required, the tree can be modified to implement all the standard dynamic tree operations, such as *find min* and *add cost*, in time $O(\log n)$, while still allowing fast linking. Such a tree would be suitable for any tree-based algorithm in which a recurrence relation similar to that of the bridge-block or block algorithm arises.

Our data structure can easily be extended to return other information about the blocks or bridge-blocks, such as an edge or vertex of maximum weight, with no loss in efficiency. In general, we can maintain any data that can be updated in constant time when two blocks or bridge-blocks are combined into one. If we are willing to increase the space used to $O(m)$, we can also list the edges or vertices in a block or bridge-block in time $O(\alpha(m, n) + k)$, where k is the number of items

listed. Our lower-bound technique can be used to show an $\Omega(\alpha(m, n))$ bound on the amortized cost per operation in any analogous on-line k-connectivity problem, assuming the graph is initially empty.

In recent work other researchers have addressed several problems that we posed in the preliminary version of this paper [25]. Tamassia and Di Battista [3] give a data structure that uses our condensible nodes and maintains the triconnected components of a graph; m operations require $O(m\alpha(m, n))$ time if the graph is initially biconnected, and $O(m \log n)$ time otherwise. Galil and Italiano (private communication, 1990) use our condensible nodes in designing a data structure to maintain the 3-edge-connected components of an initially connected graph in $O(m\alpha(m, n))$ total time. La Poutré *et al.* [12], working independently, have designed a different data structure for maintaining bridge-blocks and 3-edge-connected components that runs in total time $O(m\alpha(m, n))$. Their approach can be extended to the problem of maintaining biconnected and triconnected components.

References

[1] B. Awerbuch and Y. Shiloach. New connectivity and msf algorithms for shuffle-exchange network and PRAM. *IEEE Trans. Comput.*, **36**:1258–1263, 1987.

[2] G. D. Battista and R. Tamassia. Incremental planarity testing. *Proc. 30th IEEE Symposium on Foundations of Computer Science*, pages 436–441, 1989.

[3] G. D. Battista and R. Tamassia. On-line graph algorithms with spqr-trees. *Proc. 17th Internat. Conf. on Automata, Languages, and Programming (ICALP 1990)*. Lecture Notes in Computer Science, vol. 443, pages 598–611. Springer-Verlag, Berlin, 1990.

[4] N. Blum. On the single-operation worst-case time complexity of the disjoint set union problem. *SIAM J. Comput.*, **15**:1021–1024, 1986.

[5] G. A. Cheston. Incremental Algorithms in Graph Theory. Ph.D. thesis, Dept. of Computer Science, University of Toronto, 1976. Technical Report No. 91.

[6] S. Even and Y. Shiloach. An on-line edge deletion problem. *J. Assoc. Comput. Mach.*, **28**:1–4, 1981.

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

[8] M. L. Fredman and M. E. Saks. The cell probe complexity of dynamic data structures. *Proc. 21st ACM Symposium on Theory of Computing*, pages 345–354, Seattle, WA, May 1989.

[9] J. Hopcroft and R. E. Tarjan. Algorithm 447: Efficient algorithms for graph manipulation. *Comm. ACM*, **16**:372–378, 1973.

[10] R. Karp and V. Ramachandran. Parallel Algorithms for Shared Memory Machines. *Handbook of Theoretical Computer Science*, Elsevier, Amsterdam, 1990, pages 869–942.

[11] J. A. La Poutré. Lower bounds for the union-find and split-find problem on pointer machines. *Proc. 22nd ACM Symposium on Theory of Computing*, pages 34–44, 1990.

[12] J. A. La Poutré, J. van Leeuwen, and M. H. Overmars. Maintenance of 2- and 3-Connected Components of Graphs, Part I: 2- and 3-Edge-Connected Components. Technical Report RUU-CS-90-26, Utrecht University, 1990.

[13] J. H. Reif. A topological approach to dynamic graph connectivity. *Inform. Process. Lett.*, **25**:65–70, 1987.

[14] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *J. Comput. System Sci.*, **26**:362–391, 1983.

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

[16] R. Tamassia. A dynamic data structure for planar graph embedding. *Proc. 15th Internat. Conf. on Automata, Languages, and Programming (ICALP 1988)*. Lecture Notes in Computer Science, vol. 317, pages 576–590. Springer-Verlag, Berlin, 1988.

[17] R. Tamassia. Dynamic Data Structures for Two-Dimensional Searching. Ph.D. thesis, Co-ordinated Science Laboratory, University of Illinois at Urbana-Champagne, 1988. Technical Report ACT-100.

[18] R. E. Tarjan. Depth first search and linear graph algorithms. *SIAM J. Comput.*, **1**:146–160, 1972.

[19] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *J. Assoc. Comput. Mach.*, **22**:215–225, 1975.

[20] R. E. Tarjan. A class of algorithms which require nonlinear time to maintain disjoint sets. *J. Comput. System Sci.*, **18**:110–127, 1979.

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

[22] R. E. Tarjan. Amortized computational complexity. *SIAM J. Algebraic Discrete Methods*, **6**:306–318, 1985.

[23] R. E. Tarjan and J. van Leeuwen. Worst-case analysis of set union algorithms. *J. Assoc. Comput. Mach.*, **31**:245–281, 1984.

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

[25] J. Westbrook and R. E. Tarjan. Maintaining Bridge-Connected and Biconnected Components On-Line. Technical Report CS-TR-228-89, Dept. of Computer Science, Princeton University, Princeton, NJ, 1989.

[26] A. C. Yao. Should tables be sorted? *J. Assoc. Comput. Mach.*, **28**:615–628, 1981.