Experimental Evaluation of Algorithms for Incremental Graph Connectivity and Biconnectivity*

Madhukar Korupolu Ramgopal Mettu Vijaya Ramachandran Yuke Zhao†

Department of Computer Sciences
University of Texas at Austin, Austin TX 78712

Abstract. We consider an algorithm to maintain the connected components and the biconnected components of a graph where vertex and edge insertions are allowed. Algorithms for this problem can be applied to task decomposition in engineering design. Connected components are maintained using a disjoint set data structure and the biconnected components are maintained by a block forest. We develop a special disjoint set structure that allows selective deletion of elements, and enables us to create and use *condensible nodes*. The algorithm runs in $O(n\log n + m)$ time, where n is the number of vertices and m is the number of operations. Finally, we present extensive timing information for our implementation.

1. Introduction

We consider the problem of maintaining graph connectivity and biconnectivity information incrementally, i.e., where vertex and edge insertions are allowed. Algorithms for this problem can be applied to task decomposition in engineering design. Here we specify an implementation design that uses the algorithm presented in Westbrook and Tarjan [1] which 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 variation of the standard disjoint set structure that allows the deletion of selected elements. It is used to maintain the connected components and allows the creation and use of *condensible nodes* in the block forest. These *condensible nodes* make efficient block tree operations possible due to fast updates of their children set and parents in their tree.

Westbrook and Tarjan also describe another algorithm that uses dynamic trees to maintain the block forest, which runs in $O(m\,\alpha(m, n))$ time. For the data sets we consider, the first algorithm would likely be faster in practice due to a smaller constant factor in the running time. Hence it is the one that we describe in this paper.

Throughout this paper, n refers to the total number of vertices created; m refers to the total number of operations (*new_vertex*, *insert_edge*, and *find_block*); k refers to the total number of *find* and *merge* operations performed on disjoint sets. In section 2, we present the standard disjoint set data structure and a modified version which allows selected deletions. In sections 3 and 4, we present the details of the data structures, functions and algorithms we implemented to maintain connected components and biconnected components, respectively. In section 5 we specify the exported functions (*new_vertex*, *insert_edge*, and *find_block*) that can be considered as the user interface for maintaining connectivity and biconnectivity information incrementally; to do so, we use

* This research was supported in part by NSF grant CCR-90-23059, Texas Advanced Research Projects Grant 003658386, and Structural Dynamics Research Corporation.

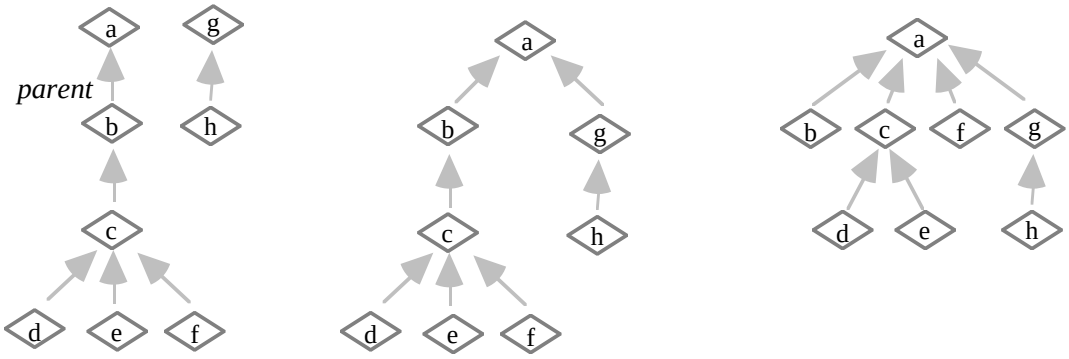
† Email addresses: madhukar@cs.utexas.edu, ramgopal@cs.utexas.edu, vlr@cs.utexas.edu, and yuke@cs.utexas.edu.

the data structures and functions specified in Sections 2, 3, and 4. Finally in section 6, we present extensive testing results on our implementation.

2. A Disjoint Set Data Structure

Disjoint sets are used to maintain both connected components and the children of nodes in the block forest in our implementation. We first present the standard disjoint set structure which supports *find* and *merge* operations [2]. Then we present a variation of the disjoint set data structure that allows selective deletion of elements. The reason this feature is needed will become clear as we proceed.

2.1 The Standard Disjoint Set Structure



(i) disjoint sets (ii) after ds_merge(a, g) (iii) after ds_find(f)

Figure 1. Disjoint set structure

A disjoint set structure is a group of *x* objects such that each of them is a member of exactly one set at any given time. Each set has a unique label which identifies it. Initially the *x* objects are in *x* different sets, each containing exactly one object. In order to handle set unions, we represent each disjoint set using a tree, where each node in the tree represents a distinct object. The trees are bottom-up trees, which means each node knows its parent but not its children. If a node's parent pointer points to itself, it is the root of that tree. The root of each tree also serves as the label of the set that particular tree represents.

The standard disjoint set structure supports the following operations:

- given some object, we find which set contains it, and return the label of set;
- given two different labels, we merge the contents of the two corresponding sets, and choose a label for the combined set.

To support these operations as well as some customized features needed by the usage, each object in the disjoint set structure must have the following fields:

- *parent:* a pointer to the parent of the node in forest representation.
- *rank* or *size*: *rank* is the upper bound of the height of the current tree, whereas *size* is the number of elements in the tree.

It is interesting to note that the use of either *size* or *rank* along with path compression yields an optimal running time of $O(k\alpha(k, n))$ for k *find* and *merge*

operations on sets with a total of *n* elements [6]. We show in Section 4.2.2 that *size* must be used for block tree eversion.

The following functions are supported as the encapsulation of the disjoint set data structure (see Figure 1):

ds_new(): create a new object as the root node and the only node of a new tree with *size* of one, return the object.
ds_find(*obj*): trace for the root of *obj* in its tree, then relink *obj* and all of its ancestors to point to the root if they don't point to it already, return the root as the label of the set.
ds_merge(*obj1*, *obj2*): if one of the two inputs is an empty set, return the other one, otherwise both *obj1* and *obj2* must be roots in their corresponding trees. If the two trees have the same *size*, let any one of the two point to the other, which will be the root of the combined tree. If the two trees differs in *size*, let the one with smaller *size* point to the one with larger *size*, which will be the new root. The size of the combined tree is *obj1.size*+*obj2.size*. Return the combined set/tree.

Although the work done by *find* to relink elements to point to the root seems excessive, it has been proven that the work done reduces the cost of future *find* operations [2]. In addition, the use of *size* to merge trees bounds the height of the tree to log*x*, where *x* is the number of elements in the set. Thus, the path compression performed by the *find* operation and the use of *size* for merging two disjoint set trees allows a running time of $O(k\,\alpha(k, x))$ as described in [2].

2.2 A Disjoint Set Structure with Selected Deletions

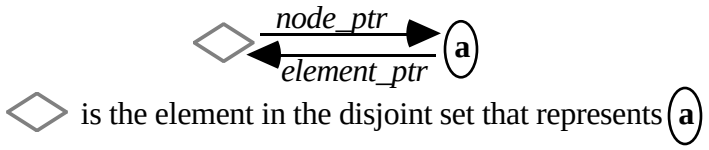


Figure 2. Additional fields

In our block tree data structure (see Section 4), each object in the set doubly links to the child node that it represents. This representation of the sets of nodes is of crucial importance, because besides *find* and *merge*, we need to delete elements from children sets when the parent-child relationships change with *block_evert* and *block_condense* operations in some block trees. This representation allows efficient manipulation of the children of a given node. The following fields are added to the elements of our disjoint set data structure (see Figure 2):

- *node_ptr*: the pointer to the block tree node which the current object represents in the disjoint set.
- *parent_node*: a pointer to the node which contains the children set (active only in the root of the disjoint set).

Using the above representation, to remove the child *c* of a node *v*, we would simply invalidate the *node_ptr* field of the element which points to *c*, and save a pointer to that element (see Figure 3). Now if we wanted to add a child to that set, instead of creating a new element, we could make the *node_ptr* field of that element point to the new child. In this way, the "holes" that are created by removing elements are reused so that the sizes of

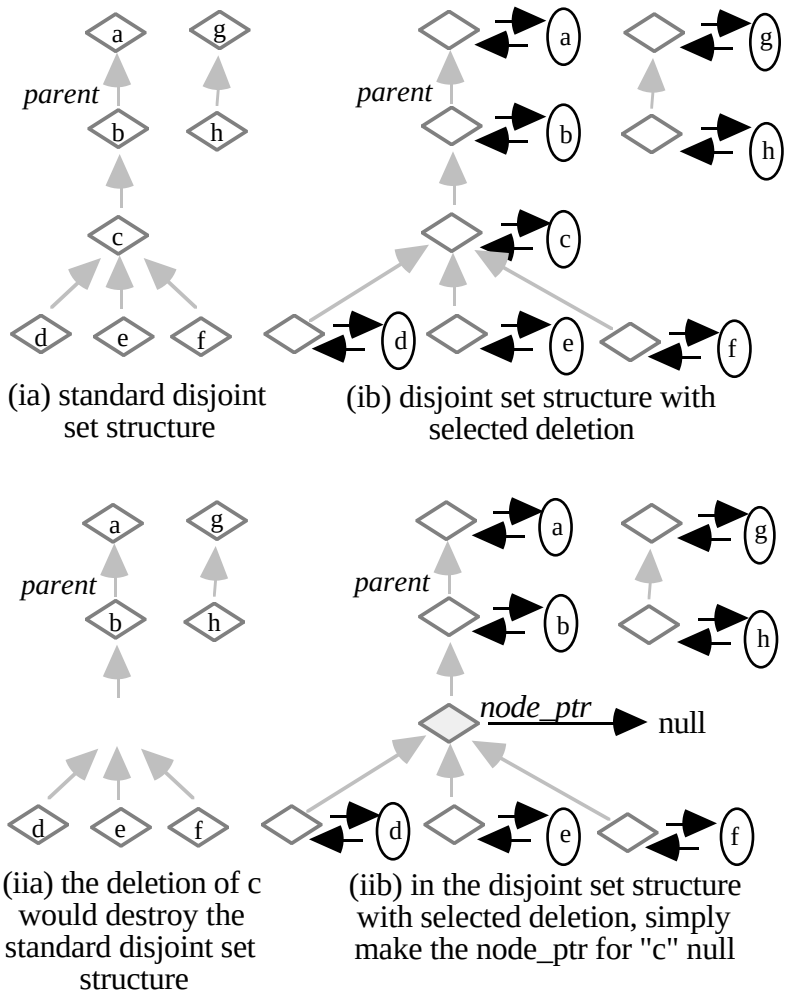


Figure 3. Disjoint set structure with selected deletions

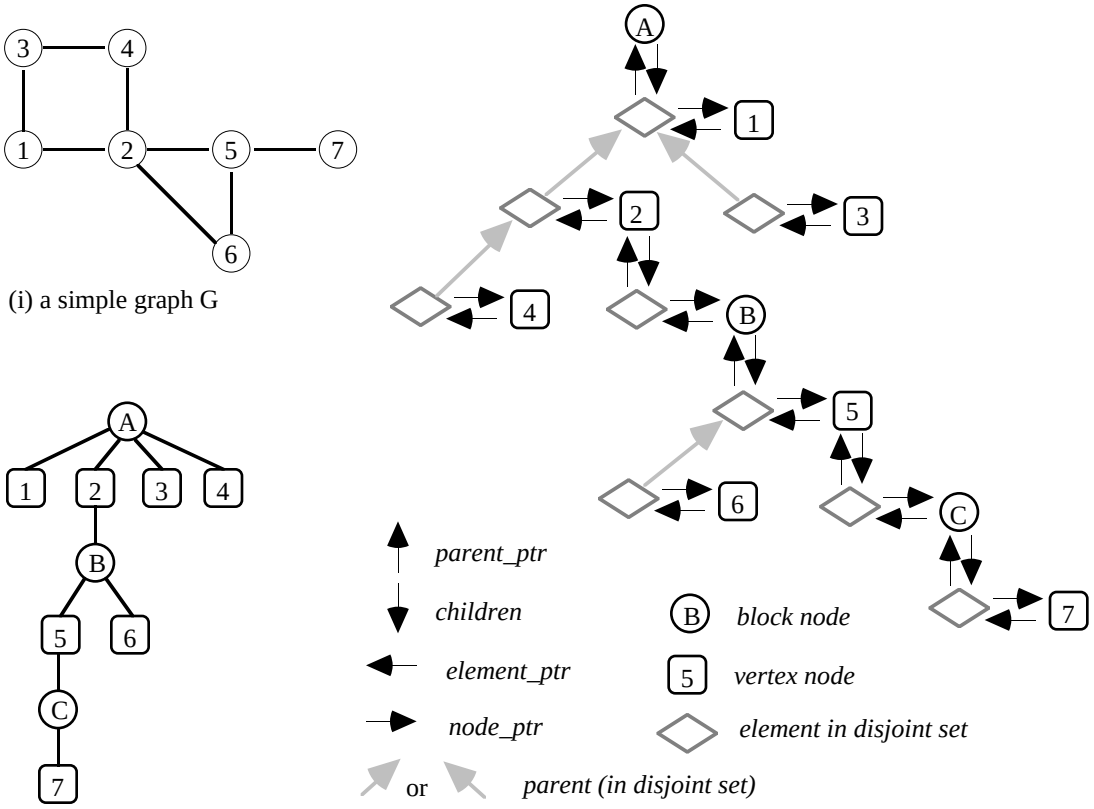
the disjoint sets are kept small enough to preserve the time bounds (see Section 5). Remaining "holes" are left in the disjoint set structure unless it represented the last child of a node. These "holes" are not removed otherwise, since they do not affect the asymptotic running time of the algorithm and their removal is very time consuming.

3. Maintaining Connected Components On-line

The data structure used to maintain connected components is exactly the disjoint set structure detailed in Section 2. The functions used to represent, update and answer queries of connected components have a clear one-on-one correspondence with the disjoint set implementation:

cc_new_vertex(): create a new vertex and put it in a new set as a new connected component, return the new vertex.
cc_find(*vertex*): given a vertex, return its disjoint set name and do a path compression along the way.
cc_merge(*set1*, *set2*): merge the two disjoint sets into one, and return the combined set.

4. A Block Tree Data Structure



(i) a simple graph G

(ii) block tree T of G

(iii) representation of T in our data structure

Figure 4. Block tree data structure
Each block node or vertex node may have two diamonds doubly linked to it; the diamond directly under it is the root of the disjoint set that represents its children; the one on its immediate left represents itself in the children set of its parent (i.e. the disjoint set that represents its siblings).

In order to maintain the biconnected components of a graph incrementally, we make use of a tree structure of the blocks and vertices of a connected graph called the *block tree* [1]. The collection of block trees given by the components of a graph $G = (V, E)$ is called the *block forest*. The block tree has two types of nodes: vertex (square) nodes, which represent the vertices of G; and block (round) nodes, which represent the blocks. The children of the block nodes represent the vertices that are contained in that block. If a vertex belongs to more than one block, it will be the child of one block node and have the other block nodes as its children. Such a vertex is called a *cut vertex* since its removal disconnects a component into two or more components.

We make use of the modified disjoint set structure of Section 2.2 to maintain the children of the block tree nodes, since we have to deal with deletions in the disjoint sets, as described in Sections 4.2.2 and 4.2.4. Each node in the data structure for the block tree represents a node in the block tree, where the children set of a node contains the children of that node in the block tree. Children sets are maintained using the disjoint set data structure, so that merging the children of two nodes can be done in $O(1)$ time. Each node

is doubly linked to its children set, that is, the node contains a pointer to the root of its children set, and that root in turn contains a pointer back to the node. (see Figure 4)

All terms such as *parent, child, siblings* and *grandparent* that appear hereafter refer to the relationships within the block forest, unless otherwise specified. Hence all vertex nodes have parent and children of block node type only, and vice versa; also, all siblings are of the same node type.

4.1 Organization of the Block Forest

Each node in our block forest contains a pointer to its representative element in its disjoint set, a set of children nodes, as well as some additional information. The set of children is implemented using the disjoint set structure presented in Section 2.2. Note that with the exception of the root, each node's parent also keeps its children using the disjoint set structure, therefore we also need a pointer in each node that points to the corresponding element in that disjoint set. Altogether, each node contains the following:

- *type*: the two types of nodes in the block tree are block nodes and vertex nodes (a vertex node is a cut vertex iff it has a non-empty children set).
- *element_ptr*: a pointer to the corresponding element in the disjoint set of children kept by its parent.
- *children*: a pointer to a set of children nodes; the root of the set also contains a pointer back to the node.
- *child_cnt*: the number of children, which is used to determine when to eliminate blocks.
- *visited*: used to mark and unmark a node while traversing a block tree.

4.2 Functions for the Block Tree

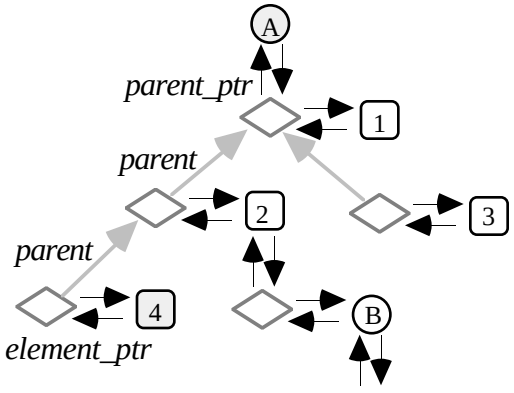


Figure 5. *block_parent*(4) = A

The above data structure was designed to support several functions which help maintain the block tree. The algorithms for the last four functions are described in detail in the following subsections. When referring to these functions, a running time of $O(f(x))$ means that an operation takes $f(x)$ pointer steps, where one *pointer step* is the amortized time taken by a *find* or *merge* operation in the disjoint set structure given in Section 2.2. These operations are:

block_new(*u*: vertex node): create a new block for the vertex which will be the root and only element of the disjoint set representing the vertices of this block; return a pointer to the new block. Running time: $O(1)$.

block_parent(*u*: vertex or block node): access the object in the disjoint set which represents the children of the parent of *u*; traverse this object to the root of the disjoint set; access and return the parent (see Figure 5). If the node is a root node, return *null*. Running time: $O(1)$.

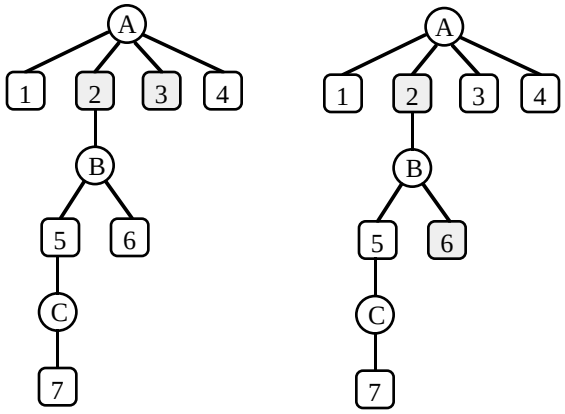
block_find(*u, v*: vertex node): given two vertices *u* and *v*, if both are in the same block return that block, return *null* otherwise. Running time: $O(1)$.

block_evert(*v*: vertex node): given a vertex node *v*, traverse the tree from *v* to the root, reversing parent pointers along the way. Running time: $O(P)$, where P is the length of the path from *v* to the root node.

block_link(*u, v*: vertex node): combine the two block trees containing *u* and *v*, where *v* is the root of the smaller tree. Running time: $O(1)$.

block_condense(*u, v*: vertex node): given two vertices, condense all block nodes in the path from *u* to *v* into a single block node. Running time: $O(P)$, where P is the length of the path between *u* and *v*.

4.2.1 Finding Blocks



(i) siblings (ii) grandparent and grandchild

Figure 6. Vertices in the same block

The *find_block* operation is used to answer the user queries and also to determine how to change the block tree upon edge insertion.

LEMMA 1. *Two vertices are in the same block iff they are siblings or one is the grandparent of the other in the block tree.*

PROOF. In the block tree structure, the parent and children nodes of any block node are vertex nodes, and they are the only vertices contained in that block node. Hence two vertices are in the same block iff either they are siblings, i.e., they have the same parent block, or if one is the grandparent of the other (see Figure 6). []

The lemma above establishes that the following procedure correctly returns the block (if any) which contains two given vertices:

```
procedure block_find(u, v: vertex node):
    { if u and v are siblings, return their parent; else if one is the grandparent of the other,
      return the block node between them; else return null }
    if block_parent(u)= block_parent(v) then
        return block_parent(u)
    else if u = block_parent(block_parent(v)) then
        return block_parent(v)
    else if v = block_parent(block_parent(u)) then
        return block_parent(u)
    else return null;
end. { block_find }
```

Since *block_parent* takes one pointer step, and *block_parent* takes *O(1)* pointer steps, *block_find* runs in *O*(1) pointer steps.

4.2.2 Everting a Tree

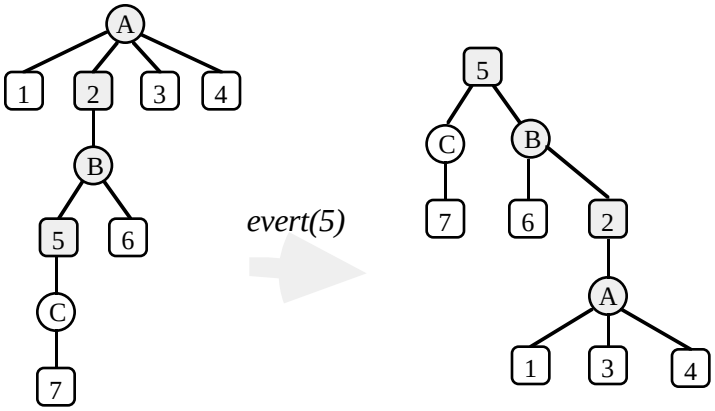


Figure 7. Eversion reverses the parent-child relationship along the path from the given vertex to the root.

In this section we specify the *block_evert* operation, which is used to re-root a block tree at a given vertex. When an edge is inserted into the graph, it may link two connected components together. In this case, the two connected components are also represented as two distinct block trees in the block forest, where the trees also need to be combined. To do so, we must re-root one of the trees at one endpoint of the new edge, then make it a child of the other vertex in the other block tree.

In order to re-root a tree at a given node *x*, for every node in the path from *x* to the root, we must reverse the parent-child relationship (see Figure 7). To do so in our data structure, for every node *u* along the path from *x* to the root, we must remove *u* from *block_parent(u).children*, and then add *block_parent(u)* to *u.children*. Note that when removing a child node from a disjoint set, it is very time consuming to update the structure of the disjoint set, so we simply change *node_ptr* to *null* and thereby invalidate one of the elements in that disjoint set; we call such elements "holes". To control the total number of "holes" generated, *block_evert* reuses them in the following way. To make use of this "hole" *h* created by the removal of a child of *u* , when adding the original block parent *B* of *u* to *u.children*, we let *h.node_ptr = B*. We cannot perform this optimization for the root node, since it has no parent and we only remove a child, hence that "hole" remains in the data structure. (see Figure 8)

Given a node *x* which is to be the new root of the block tree, the following procedure describes how the block tree data structure is re-rooted at *x*:

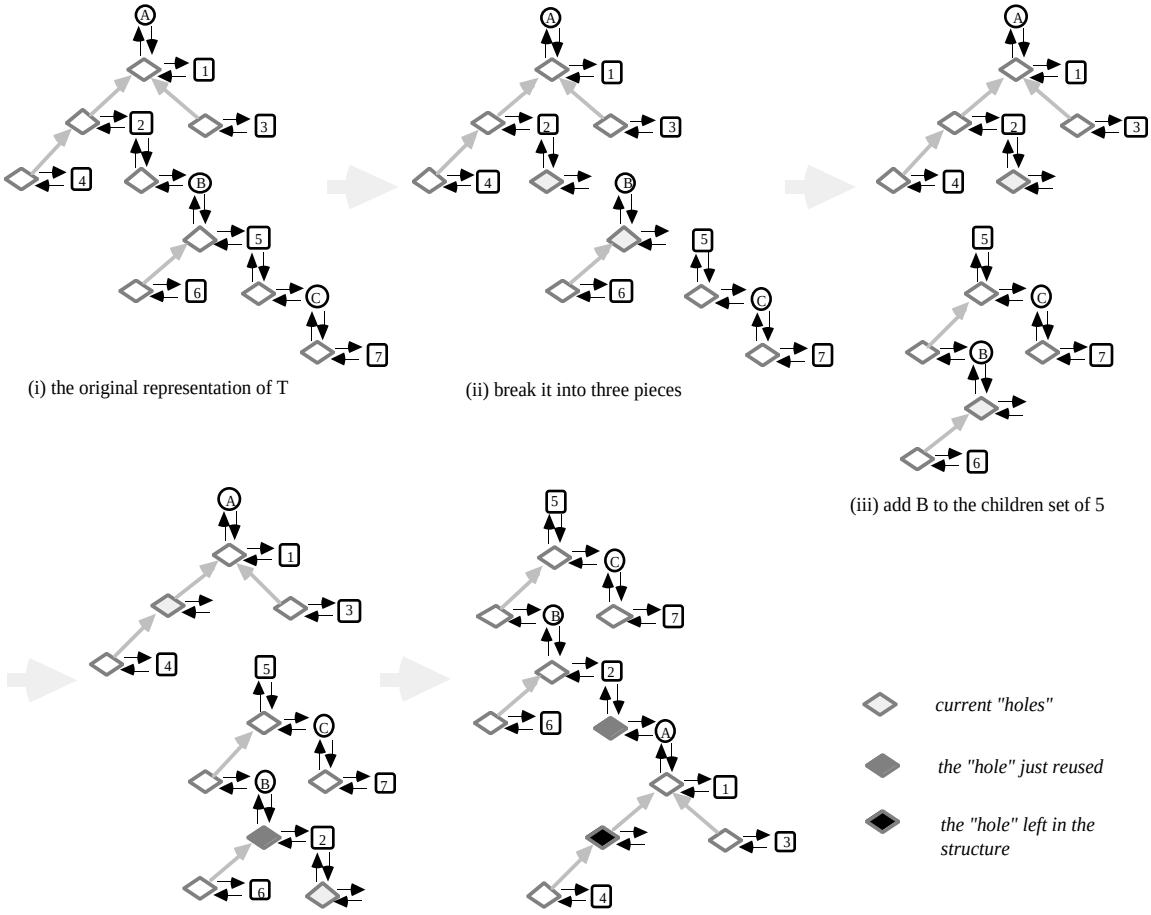


Figure 8. *block_evert*(5)

procedure *block_evert(x: vertex node)*

{ initialize pointers to process first node }
curr := x;
parent := block_parent(curr);
grandparent := block_parent(parent);

{ process the first node, *x*, separately since it will not be put in a "hole", then save the first "hole" and sever the node *curr* from its element in its disjoint set }
hole1 := curr.element_ptr;
(curr.element_ptr) →node_ptr := null;
curr.element_ptr := null;

{ save the second "hole" sever the parent node from its element in its disjoint set }
hole2 := parent.element_ptr;
(parent.element_ptr) →node_ptr := null;
parent.element_ptr := null;
add parent to curr.children

{ process remaining nodes by using *hole1* as the first "hole" to be filled, and *hole2* as the "hole" to be used for the parent node }
while grandparent ≠ null do

```
        { update the curr, parent and grandparent pointers }
        curr := parent;
        parent := grandparent;
        grandparent := block_parent(parent);

        { sever parent from its disjoint set }
        if grandparent ≠ null then
            (parent.element_ptr) →node_ptr := null;

        { insert the parent into curr.children by filling hole1, and make hole1 point to
          parent's element in the disjoint set, which is the new "hole" in the set }
        swap(hole1, parent.element_ptr);
        (parent.element_ptr) →node_ptr := parent;

        { swap hole1 and hole2 since hole1 should contain the more recent "hole" }
        swap(hole2, hole1);

    end { while }

    { decrement the children count of the original root }
    parent.child_cnt := parent.child_cnt - 1;
    { if the original root node now has zero children, then eliminate that block }
    if parent.child_cnt = 0 then
        block_parent(parent).child_cnt := block_parent(parent).child_cnt - 1;
        (parent.element_ptr) →node_ptr := null;
        free storage used by parent node
end. { block_evert }
```

This algorithm will produce one "hole" (saved in the variable *hole1*) per *block_evert* operation. It is important to recognize that the number of "holes" generated by eversion could not have been bounded by the number of vertices without the mechanism that fills the parent of a node into the "hole" left by its child. This optimization preserves the performance of the disjoint set structure by preventing any significant increase in the size of these sets. More importantly, this optimization could not have been performed without the data structure described above. The modified disjoint set structure is maintained even when a node is removed from the set since the elements act as virtual copies of the actual nodes. It is this abstraction that makes the existence of the "holes", thus their future use, possible.

The running time of *block_evert* is linear in the length of the path; since the number of vertex nodes in a block tree is at least the number of block nodes, *block_evert* takes at most $O(k)$ pointer steps in the block tree of k vertices. By defining the size of a block tree to be the same as the size of its corresponding connected component, Westbrook and Tarjan [1] show that everting the smaller block tree ensures $O(n \log n)$ performance. This and the lemma below justify the use of *size* instead of *rank* in our modified disjoint set data structure.

LEMMA 2. *The connected component with smaller rank does not necessarily have smaller size.*

PROOF. We establish the claim by a counter-example. If we start with a singleton set and only merge singleton sets to it, the *rank* will always remain 2 for the resulting set, since the first merge increases the *rank* by one and subsequent merges do not. We can merge singleton sets until we get a set S of *rank* 2 with k elements for any k. Then, we can obtain

another set *T* of *rank* 3 which contains only 4 elements by merging two pairs of singleton sets and then merging these pairs together. Above, we've shown that although *S.rank*<*T.rank* , we know $|S| > |T|$ for *k*>*4*. Therefore a disjoint set with smaller *rank* in the disjoint set structure does not guarantee a smaller *size.* []

4.2.3 Linking Two Trees

When we insert an edge between two connected components, we evert the smaller block tree; after the eversion is done as in Section 4.2.2, we link it to the bigger block tree.

The function *block_link* takes two vertex nodes, *u* and *v*, as its arguments, where *u* is the endpoint of the new edge that resides in the bigger block tree and *v* is the end point of the edge that resides in the smaller block tree. At this point the eversion has already been done on *v*, hence *v* is the root (temporary) of the smaller block tree. Also notice that in the block forest, *insert_edge*(*u, v*) creates a new block where the only vertices contained in it are *u* and *v*. Hence in order to link the two block trees together, we need to create a new block with *v* as its only child and make *u* its parent:

procedure *block_link(u, v: vertex node)*
    B := block_new(v);
    add B to the children set of u
end. { *block_link* }

Adding a block to a children set is merely a merge of two disjoint sets, which runs in a constant time, plus some pointer manipulations. Since *block_new* also takes a constant time, *block_link* runs in *O*(1) time.

4.2.4 Condensing a Path

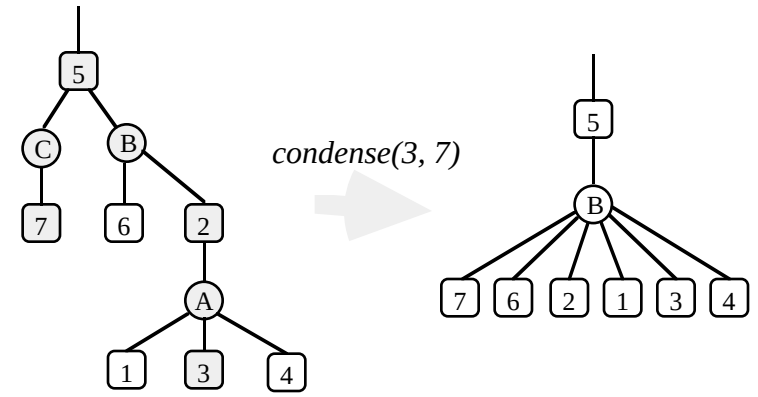


Figure 9. Condensation collapses all the blocks along the path between the two vertices into a single block.

Another possible scenario of inserting a new edge occurs when the new edge links two vertices that are in the same connected component but different blocks. Its effect on the block tree is that all the block nodes along the path between them collapse into a single block node, while all the vertex nodes contained in these blocks and along the path will belong to the new block node. (see Figure 9)

In order to condense these blocks, we first introduce two subroutines: *findlca* and *condense_path*. The *findlca* algorithm proceeds by walking up the tree simultaneously

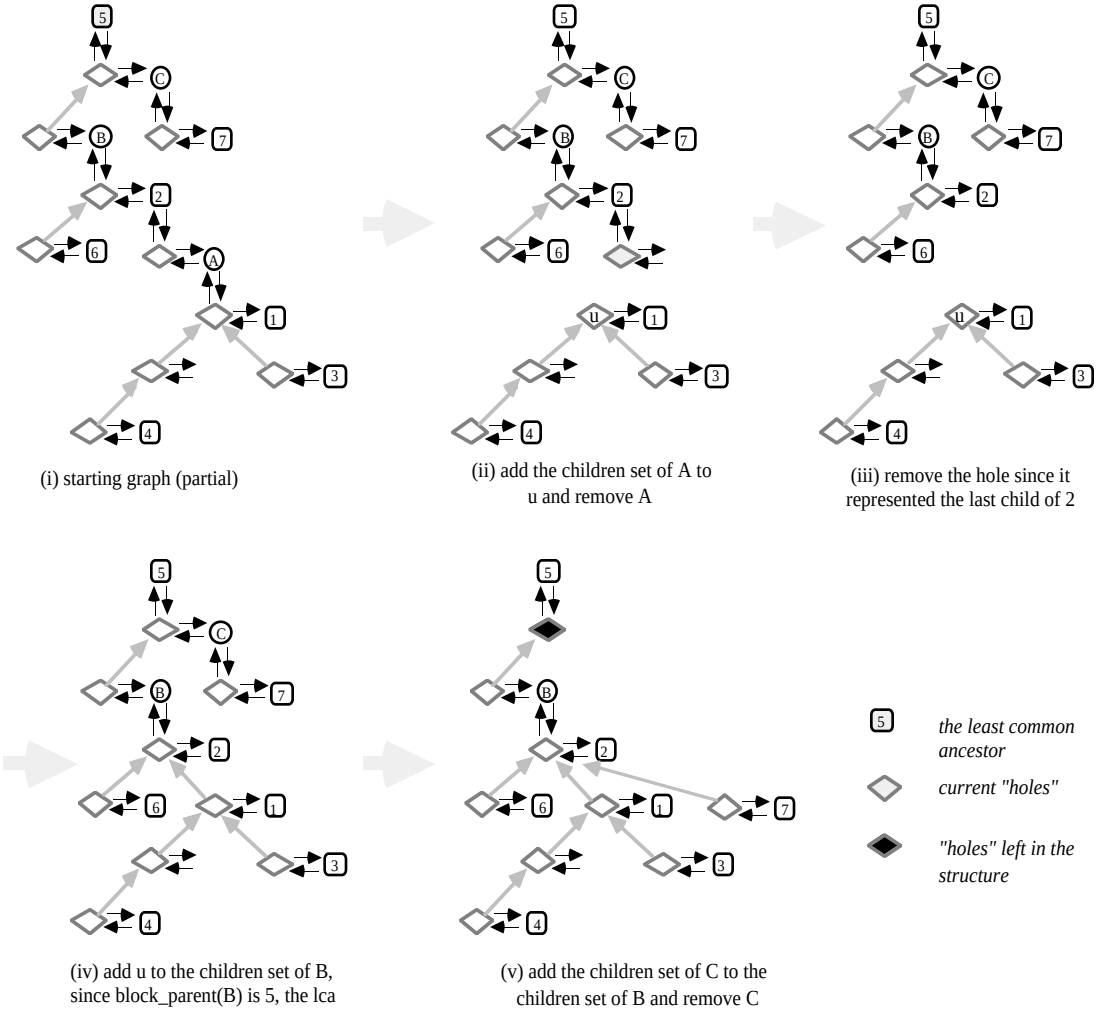


Figure 10. *block_condense*(3, 7)

from its two vertex node arguments, until the paths intersect at their least common ancestor, which is returned by the function:

subroutine *findlca(v1, v2: vertex node): vertex or block node*
{ find the least common ancestor of *v1* and *v2* by walking up the tree simultaneously from both of them, until the paths intersect at their lca }
lca := v1;
save := v2;
while lca.visited = false
lca.visited := true;
if lca is the root of block tree then
{ ignore the path that has reached the root from now on }
lca := save;
save := block_parent(save);
else
{ advance to the next node on the current path, and switch to the other path }
tmp := block_parent(lca);
lca := save;
save := tmp;
unmark all the nodes that were marked;

```
    return lca;
end. { findlca }
```

The function *condense_path* takes two arguments, *x* and *y*, where *x* is a vertex node and *y* is an ancestor node of *x*. If *y* is a block node, all of the block nodes along the path from *x* to *y* are combined into *y*; if *y* is a vertex node, they are combined into the child block node of *y* which is on the path. The intermediate blocks along the path are deleted, and their children are merged to the children set of the final block. This can be done in our data structure since each block has a pointer to the root of its children set. The deletion of a block is done by invalidating the *node_ptr* field of the element which represents this block in the disjoint set of its siblings. Hence each deletion of a block may create one "hole".

The difference between these "holes" and the "holes" produced by eversions (see Section 4.2.2) is that the "holes" created by eversions were used to represent vertices while the "holes" generated by path condensations were used to represent blocks.

```
subroutine condense_path(x: vertex node, y: vertex or block node): block node
    { start with the parent block of x, initialize }
    b := block_parent(x);
    v := block_parent(b);
    u := empty set;
    { process the blocks along the path one by one, until we reach the lca if it is a block,
      or until its child if it is vertex }
    while b ≠ y and v ≠ y
        add the children set of b to u;
        remove b from the children set of its parent, v;
        if v.child_cnt = 0 then make v.chidren an empty set;
        free(b);
        b := block_parent(v);
        v := block_parent(b);
    add u into the children set of b;
    return b;
end. { condense_path }
```

Two vertex nodes that are in the same connected component and different blocks are given as the arguments to *block_condense*. Using *findlca* and *condense_path*, the algorithm is the following (also see Figure 10):

```
procedure block_condense(v1, v2: vertex node)
    lca := findlca(v1, v2);
    { condense along the paths to lca, the least common ancestor }
    if lca = one of the two vertices
        condense_path(the other vertex, lca);
    else
        b1 := condense_path(v1, lca);
        b2 := condense_path(v2, lca);
        if b1 = b2 then          { b1 = b2 = lca, which is a block node }
            do nothing;
        else                     { b1 and b2 are children of lca, a vertex }
            add the children set of b2 to the children set of b1;
            remove b2 from the children set of its parent, lca;
            free(b2);
end. { block_condense }
```

Subroutine *findlca* takes $O(P)$ pointer steps, where P is the length of the path between the two given vertices (see *findpath*, [1]), and *condense_path* is a sequence of pointer operations along the path, hence it runs in $O(P)$ pointer steps as well. All other operations involved in *block_condense* are merely pointer manipulations, therefore *block_condense* also runs in $O(P)$ pointer steps.

5. Exported Functions to Maintain Connectivity and Biconnectivity

The data structures presented above can independently maintain graph connectivity and biconnectivity properties incrementally. To combine these operations, we will describe a set of functions: *new_vertex*, *find_block*, and *insert_edge*, which update both connectivity and biconnectivity when called. The functions are as follows:

1) *new_vertex():*
create a new vertex and a new block which contains it.

2) *find_block(u, v: vertex)*:
call *block_find(u, v)* and return the result.

3) *insert_edge(u, v: vertex)*:
if *u* and *v* are in the same block, do nothing.
otherwise if they are in the same connected component, call *block_condense(u, v)*
otherwise they are in different connected components, in which case we combine the two block trees by calling *block_evert()* with the vertex in the smaller tree, and make the root of the resulting tree a child of the vertex in the larger tree. Then merge the two connected components by calling *cc_merge()*.

Both *cc_new_vertex* and *block_new* take $O(1)$ time, so *new_vertex* also runs in $O(1)$ time. Since *find_block* is merely a call to *block_find*, *find_block* also runs in $O(1)$ pointer steps. In order to analyze the running time for *insert_edge*, we need to prove the following lemmas:

LEMMA 3. *In any sequence of insert_edge operations, there can be at most n-1 "holes" caused by eversions in a graph of n vertices.*

PROOF. After each eversion, two block trees are combined into one, hence for a graph of *n* vertices, there can be at most *n-1* evert operations before all *n* vertices are combined into a single block. Since each eversion creates at most one "hole" in the children set of the root node, there can be at most *n-1* such "holes" in the block forest. []

LEMMA 4. *In a graph of n vertices, the sum of the number of block nodes and the number of "holes" generated by path condensations is never more than n .*

PROOF. A "hole" can only be generated during path condensation when a block is absorbed by another block. This causes the resulting block to contain all the vertices of the initial blocks. Since every "hole" replaces a previous block node, the sum of the number of block nodes and the number of "holes" generated by path condensations is always bounded by *n*. []

Remark. Since there is no block with an empty children set in the block forest, there can be at most *n-1* such absorptions before all *n* vertices are in a single block, i.e. the whole

graph becomes a single block. Hence there can be at most *n-1* "holes" generated by path condensations.

To calculate the running time of *find* and *merge* operations in our modified disjoint set data structure, we must first determine how many elements are in all the disjoint sets in the block forest. There will be *n* elements which represent vertex nodes in the block forest; lemma 3 shows that there will be at most *n*-1 "holes" created by eversion, and lemma 4 shows that the sum of elements representing block nodes and "holes" created by path condensations is at most *n*. Hence, there will be at most 3*n*-1 elements in all the disjoint sets in the block forest. This means the running time of *k find* and *merge* operations will be $O(k\alpha(k, 3n\text{-}1))$, and the amortized time per operation will be $O(\alpha(k, 3n\text{-}1))$.

THEOREM 1. *The data structure above can maintain the graph connectivity and biconnectivity incrementally in O(nlogn + m) time and O(n) space.*

PROOF. Tarjan and Westbrook [1] show that the total number of pointer steps taken by *block_evert* and *block_condense* is $O(n\log n)$. For a sequence of *m insert_edge*, *find_block*, and *new_vertex* operations where *m* is in $\Omega(n)$, Tarjan and Westbrook show that the total number of pointer steps required is $O(m + n\log n)$. However, this analysis does not take into account the "holes" that need to be created in the block tree data structure during *block_evert* and *block_condense* operations.

By the analysis given in [1], our data structures support a sequence of *m* operations in $O(m + n'\log n')$ time, where n' represents the total number of elements in the disjoint sets of the block tree including "holes". From Lemmas 3 and 4 it follows that $n' \leq 3n\text{-}1$, hence n' is $O(n)$. Hence, our data structures can be maintained in $O(m + n\log n)$ time. []

6. Experimental Results

To evaluate the efficiency of the algorithms above, we ran our implementation of the above algorithms and data structures with a variety of initial graphs and update/query sequences. Below we give the results of our experiment, as well as a comparison with an implementation of another set of dynamic graph algorithms.

6.1 A Note on the Implementation

With the algorithms and data structures above, we found that two equivalent implementations were possible. The first consists of managing connectivity and biconnectivity information at the exported functions level, which means that for each vertex, two pointers would be required to access all the needed information, one for the connectivity data structure and one for the biconnectivity data structure. The second implementation consists of maintaining one pointer per vertex; this vertex would point to a node in the block tree, from which connectivity information can be accessed if needed. The relative differences are minor, since the first encapsulates connectivity and biconnectivity, whereas the second uses nodes in the block tree as an abstraction for a vertex in the graph. We report the test results obtained from the second implementation.

6.2 Test Environment and Test Data

vertices	edges	updates/ queries	init. time	std. deviation	update time	std. deviation
500	250	50	0.012347	0.002972	0.001422	0.000364
500	250	500	0.012565	0.002923	0.009204	0.000419
500	250	2000	0.013404	0.002603	0.017593	0.000552
500	500	50	0.020703	0.002683	0.001074	0.000216
500	500	500	0.021065	0.002584	0.005088	0.000657
500	500	2000	0.021553	0.002477	0.010002	0.000304
1000	500	100	0.024278	0.005724	0.002576	0.000295
1000	500	1000	0.025348	0.005487	0.018880	0.001006
1000	500	5000	0.026972	0.004854	0.040065	0.001356
1000	1000	100	0.042146	0.004986	0.001933	0.000350
1000	1000	1000	0.043066	0.004651	0.010167	0.000715
1000	1000	5000	0.044012	0.004639	0.023937	0.000936
5000	2500	500	0.125514	0.024981	0.011869	0.000721
5000	2500	5000	0.131107	0.023097	0.100646	0.002215
5000	2500	30000	0.138783	0.020433	0.244817	0.006014
5000	5000	500	0.221482	0.022214	0.007709	0.000541
5000	5000	5000	0.225377	0.021231	0.054985	0.001670
5000	5000	30000	0.229629	0.020147	0.159582	0.003700
10000	5000	1000	0.257545	0.048957	0.024966	0.000917
10000	5000	10000	0.268006	0.044069	0.207359	0.006251
10000	5000	70000	0.283729	0.038906	0.546014	0.015409
10000	10000	1000	0.446950	0.043444	0.015335	0.000649
10000	10000	10000	0.454323	0.041261	0.113661	0.001467
10000	10000	70000	0.464003	0.038620	0.371572	0.004577
20000	10000	2000	0.529391	0.093537	0.049379	0.000782
20000	10000	20000	0.548906	0.086237	0.408115	0.010519
20000	10000	150000	0.586515	0.074454	1.187864	0.048198
20000	20000	2000	0.900329	0.080149	0.028736	0.000886
20000	20000	20000	0.917808	0.075086	0.226257	0.006346
20000	20000	150000	0.942790	0.068148	0.833192	0.019910

Table 1. Statistics for various input graphs and update/query sequences

Our implementations were run on a Sun SPARCstation 5/110 with a 110MHz CPU and 32 MB of RAM running the Solaris operating system. The algorithms and data structures were written in the C programming language. The UNIX library function `getrusage()` was used for all time measurements.

We tested our implementations on graphs with 500, 1000, 5000, 10000 and 20000 vertices, respectively. The initial graphs are very sparse (with exactly $n/2$ edges) or sparse (with exactly n edges) and are randomly generated. They are loaded using *new_vertex* and *insert_edge* operations. Ten such graphs are generated for each category, then 3 random test sequences of different sizes are run on each graph. The test sequences are uniformly mixed with 50% of *insert_edge* and 50% of *find_block*. The sizes of sequences are approximately $n/10$, n and $2((n\ln n)/2 - n)$, respectively. The largest sequence size was chosen as $2((n\ln n)/2 - n)$ since a random graph is very likely to become connected when the number of its edges reaches approximately $n\log n$ [4], after which every operation will be within one block tree. Once the block tree become biconnected the performance of the algorithm speeds up substantially from $\theta(n\log n + m)$ to $\theta(m\,\alpha(m, n))$ as indicated by [1]. Hence we take the length of the update/query sequence to be $n\log n/2$, minus the number of edges in the initial graph, and multiply it by two since half of the test sequence operations are *find_block* queries.

6.3 Test Results

The times (given in seconds) obtained from our tests are listed in Table 1, in ascending order of vertices and edges. We used the test results from Alberts et al. [3] as reference, which to our knowledge is the only other experimental study of dynamic graph algorithms. Alberts et al. measured their implementation of four different fully dynamic algorithms that maintained graph connectivity. For random inputs they indicated that none of these four algorithms was dominant in all cases. Our implementation of the Westbrook-Tarjan algorithm maintains both connectivity and biconnectivity incrementally. Since there was no dominant algorithm in their implementation, we make comparisons between our algorithm and their fastest algorithm for similar test data. To make the comparisons more accurate, we did not include *new_vertex* operations in the update/query sequences because Alberts et al. do not implement this operation.

Since the dynamic algorithms implemented by Alberts et al. were run on DEC 4000/720, to make comparisons using time values, we use the SPECintrate92 benchmark, which is derived from a set of integer benchmarks and estimates the throughput of a machine for integer based programs. The SPECintrate92 value for their DEC 4000/720 machine was 5144, compared to a SPECintrate92 value of 1864 for our machine [5]. Based on this benchmark, their machine is approximately 2.76 times faster than the one we used.

Since Alberts et al. measured times for graphs with at most 500 vertices, we can only make a few comparisons between their implementations and ours. The following time comparisons have been normalized according to the SPECintrate92 benchmark:

For very sparse initial graphs (*n/2* initial edges) with 500 vertices and test sequences of 500 updates/queries, our implementation ran approximately 30 times faster than their fastest algorithm.

For sparse initial graphs (*n* initial edges) with 500 vertices and test sequences of 500 update/queries, our implementation ran approximately 70 times faster. We note that as the graph becomes more dense, it is also more likely to become connected or biconnected. As this happens, fewer *block_evert* and *block_condense* operations are needed and the average time per *insert_edge* decreases.

The next closest comparison is for a sequence of 5000 updates/queries, Albert et. al. run the sequence on initial graphs of 500 vertices and 500 initial edges, while our test sample is run on initial graphs of 1000 vertices and 500 initial edges. In this case, our implementation runs approximately 93 times faster.

We can also compare our largest sample, a sparse graph with 20,000 vertices and a test sequence of 150,000 updates/queries, and their largest documented sample, an initial graph with 500 vertices, 62,500 initial edges and a test sequence of 5000 updates. For this case, our sample runs almost 4.5 times faster.

The above difference in the running times of our implementation and Alberts et al. implementations is expected theoretically, since we have shown that our implementation runs in $O(\log n)$ amortized time per update, whereas the asymptotically fastest algorithm tested by Alberts et al. runs in $O(\log^2 n)$ expected time per operation. Although our implementation maintains both connectivity and biconnectivity while the Alberts et al. maintain only connectivity, the algorithms we used were incremental, whereas the algorithms tested by Alberts et al., are fully dynamic, and therefore handle edge deletions, which are harder to deal with.

Acknowledgments. This paper is part of an undergraduate honors thesis project undertaken by Ramgopal Mettu and Yuke Zhao; the coding of the implementations described in Section 6.1 was done entirely by them. We plan to extend the results given here by studying and implementing incremental algorithms for graph triconnectivity.

References

[1] J. Westbrook and R. E. Tarjan. Maintaining Bridge-Connected and Biconnected Components On-Line. Algorithmica (1992) 7: 433-464

[2] R. E. Tarjan. Efficiency of a Good But Not Linear Set Union Algorithm. Journal of the Association for Computing Machinery, 22 (1975), 215-225.

[3] D. Alberts, G. Cattaneo, and G. F. Italiano. An Empirical Study of Dynamic Graph Algorithms. ACM-SIAM SODA (1996): 192-201.

[4] J. H. Spencer. Ten Lectures on the Probabilistic Method. Capital City Press, 1994, 2nd edition.

[5] J. Dimarco. SPEC list. http://hpwww.epfl.ch/bench/SPEC.html

[6] R. E. Tarjan and J. van Leeuwen. Worst-Case Analysis of Set Union Algorithms. Journal of the Association for Computing Machinery, 31 (1984), 245-281.

Plots of Test Results

The plots on the following page were made from the test data given in Table 1; all graphs have n vertices. The first plot shows the average times taken by update/query sequences run on initial graphs with $n/2$ edges. Each set of update/query sequences is represented in the plot with its own set of points connected by line segments. For example, the line segments labeled "$n/10$" represent the average running times of the update/query sequence with $n/10$ edges. The second plot shows the times measured from running the same size update/query sequences on initial graphs with n edges.