# Experimental Evaluation and Comparison of Algorithms for Incremental Graph Biconnectivity*

Ramgopal Mettu Yuke Zhao Vijaya Ramachandran

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

June 9, 1997

**Abstract.** We describe our implementation of 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 implemented an incremental biconnectivity algorithm presented in Westbrook and Tarjan [8] which runs in $O(n\log n + m)$ time, where $n$ is the number of vertices and $m$ is the number of operations. However, the algorithm in [8] does not address the issue of deletions in disjoint sets, which seem to be needed in order to implement the algorithm correctly. Hence we develop the concepts of *dynamic holes* and *static holes* to support variations of the standard disjoint set structure that allow the deletion of certain elements. We present four different implementations of the incremental biconnectivity algorithm which utilize these variants of the standard disjoint set structure, and analyze their performance. We present extensive timing information for each of these implementations.

## 1. Introduction

An *incremental graph algorithm* is an algorithm that maintains certain graph properties while new vertices and edges are inserted into the graph. In this paper, we consider incremental graph algorithms that maintain graph *connectivity* and *biconnectivity* information. The partitions of a graph $G$ according to these properties are called its *connected components* and *biconnected components*, respectively. Two vertices are in the same *connected component* of $G$ if and only if there is a path connecting them. A *biconnected component* is a maximal subgraph of $G$ where after the removal of any one vertex, the subgraph is still connected. Hence the biconnected components within a connected component are connected by vertices whose removal disconnect the connected component. These vertices lie in more than one biconnected components and are called *cut-vertices*.

Incremental graph algorithms that maintain connectivity and biconnectivity can be applied to task decomposition in engineering design. The need for maintaining biconnectivity incrementally also comes up in problems relating to networks, CAD/CAM, and distributed computing [8].

In this paper, we describe an incremental biconnectivity implementation design that uses an algorithm presented in Westbrook and Tarjan [8] which runs in $O(n\log n + m)$ time, where $n$ is the number of vertices and $m$ is the number of operations. In order to implement the algorithm, we found the need to delete certain elements from disjoint sets. This issue is not addressed in [8], however. Hence we developed the concepts of *dynamic holes* and *static holes* to support variations of the standard disjoint set structure that allow the deletion of certain elements. These variants of the standard disjoint set structure are used to maintain the connected components and enables efficient block tree operations in the block forest.

Westbrook and Tarjan [8] 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 is likely to 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 describe the standard version

* This research is part of an undergraduate honors thesis project undertaken by Ramgopal Mettu and Yuke Zhao and was partially supported by NSF grant CCR/GER-90-23059 and Structural Dynamics Research Corporation.

and a modified version of the disjoint set data structure and the basic block tree data structure. In Sections 3 through 6, we present four implementations of incremental biconnectivity, utilizing the two versions of disjoint set data structure and the concepts of *dynamic* and *static holes*. In Section 7, we present the experimental evaluation of the four implementations described in previous four sections. Finally, we conclude the paper with some remarks in Section 8.

## 2. Basic Data Structures for Connected Components and Block Tree

In this section, we describe two basic data structures used in our implementations, the *standard disjoint set* and *block tree* data structures. We also present the concept of a *dynamic hole* and describe how it is used in a modified version of the disjoint set data structure to allow deletions.

The basic data structure for connected components is the *disjoint set*. The data structure for biconnected components is a data structure called *block tree*. In block tree data structure, we need a disjoint set data structure that allows deletions.

In Section 2.1, we present the standard disjoint set data structure, the definition of *dynamic holes*, and a modified version of the disjoint set data structure which allows deletions. In Section 2.2, we give an overview of the data structure for the block tree. Later, in Section 5, we define *static holes* and a modified disjoint set data structure that supports deletions using static holes.

### 2.1 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 [6]. Then we present a variation of the disjoint set data structure that allows selective deletion of elements.

#### 2.1.1 Standard 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:

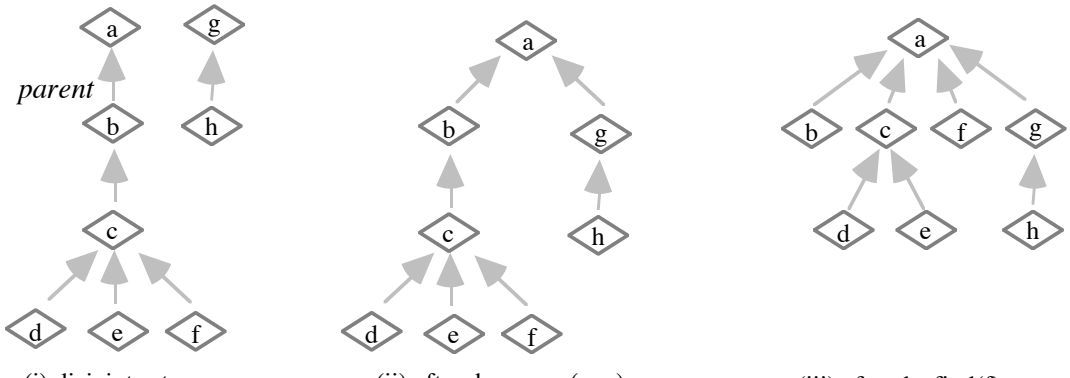



Figure 1. Disjoint set structure

- 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 an upper bound on 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 [7]. We show in Lemma 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 differ 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 [6]. 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 [6].

### 2.1.2 Dynamic holes in the Disjoint Set Data Structure

In order to manipulate the children sets of nodes in our block forest, we need to both merge two sets together, as well as delete certain elements in a given set efficiently. The disjoint set data structure was chosen to allow fast merging. In order to quickly delete elements from the disjoint set data structure, we developed the concept of *dynamic holes*.

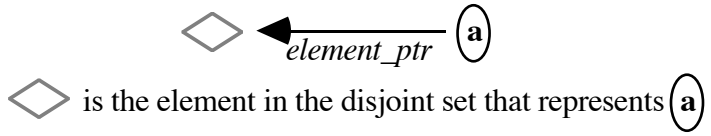



Figure 2. Additional fields

A *dynamic hole* is a node in a tree in the disjoint set data structure which does not represent any element of the set. Its only purpose is to maintain the internal tree structure of the disjoint set data structure. If we do not use such nodes, in order to delete an element in the disjoint set structure we would have to relink all of the children of that element, which would often be a very expensive operation. We describe the deletion process in the following section.

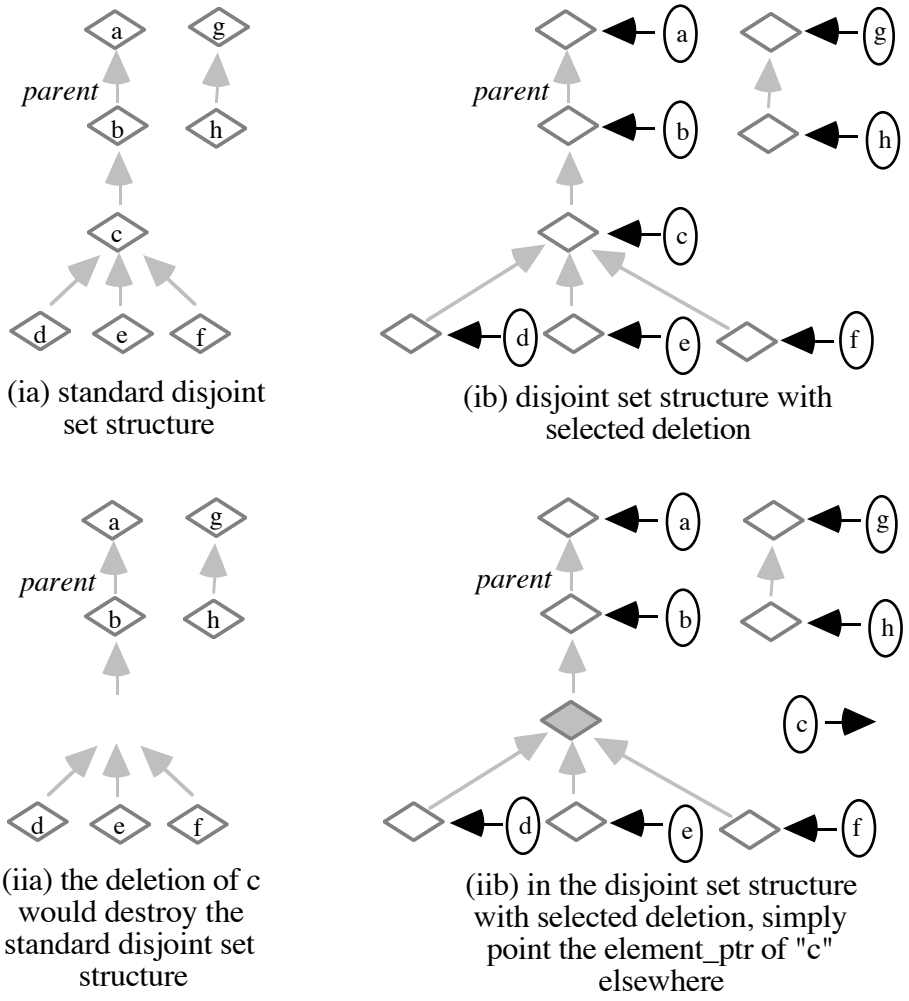


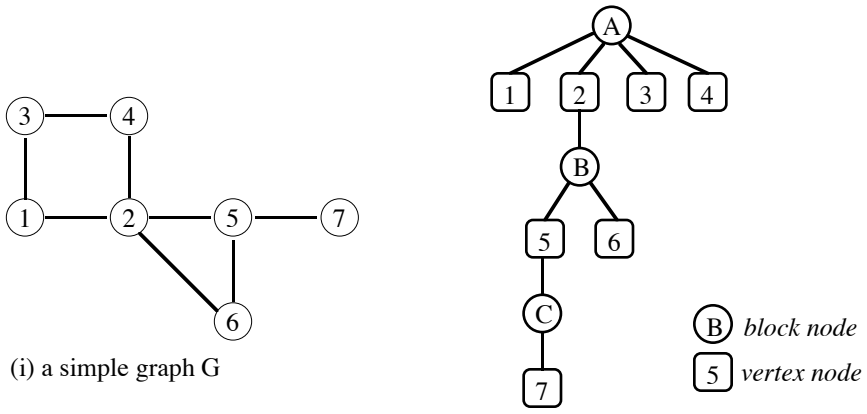

Figure 3. Disjoint set structure that allows deletions

### 2.1.3 Disjoint Set Structure that allows Deletions

We need a disjoint set data structure that allows deletions in order to maintain the children sets in the block tree data structure. This modified disjoint set data 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 *dynamic holes*, and thus their future use, possible.

Using the above representation, we delete a child $c$ from a children set of a node $v$ by making $c \rightarrow$*element_ptr* point elsewhere. The element in the children set of $v$ that used to represent $c$ is now a *dynamic hole*, since it no longer represents any child node of $v$ (see Figure 3). We may reuse this *dynamic hole* when another node is to be added to the children set of $v$, so that the sizes of the disjoint sets are kept small enough to preserve the time bounds (see Lemma 3). These *dynamic holes* are not removed, since they do not affect the asymptotic running time of the algorithm and their removal could be very time consuming.

Since we will use disjoint set structure extensively in this paper, we will often express the running time of an algorithm in terms of pointer steps. A *pointer step* is the amortized time taken by a *find* or *merge* operation in the disjoint set structure. This time bound is $\alpha(k, n)$, where $k$ in the number of *find* and *merge* operations and $n$ is the number of elements in the set.




(i) a simple graph G

(ii) block tree T of G

Figure 4. Block tree data structure

### 2.2 Block Tree Data Structure

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* [8]. 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. A *block* is equivalent to a biconnected component in the graph, where two vertices are in the same *biconnected component* if and only if there are two vertex disjoint paths between them or the component is a single edge. 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.

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.

The children set of a block node are the vertex nodes contained in that block except its parent cut-vertex. The children set of a vertex node are the blocks in which it is contained, except for its parent block. The children sets of block nodes in all of our implementations are maintained using a disjoint set data structure. In Implementation I, children sets of vertex nodes are maintained in disjoint sets as well, while direct pointers are used in the other implementations (see Section 4).

The need for the disjoint set data structure that allows deletions arises in the maintenance of the block tree data structure. Since the children sets of the block nodes are always organized in disjoint sets, changing the block to which a vertex belongs means removing it from one set and merging it into another. Hence the creation of *dynamic holes* is needed when the parent-child relationship changes within the block tree data structure.

In the next four sections, we present four different implementations of incremental biconnectivity. We classify an implementation as "homogeneous" when both block nodes and vertex nodes use disjoint sets to maintain their children sets, and "mixed" when block nodes use disjoint sets and vertex nodes use direct pointers. We also classify the implementations based on whether they use *dynamic holes* or *static holes*. *Dynamic holes* are defined in Section 2.1.2; *static holes* are created when a newly created block only has a cut vertex as its child (see Section 5). *Dynamic holes* can only be created with the modified disjoint set data structure, whereas *static holes* can be created with both the standard and modified disjoint set data structure.

In Section 3, we present a homogeneous implementation of the block forest which uses *dynamic holes*. In Section 4, we present the second implementation, which uses a mixed block tree

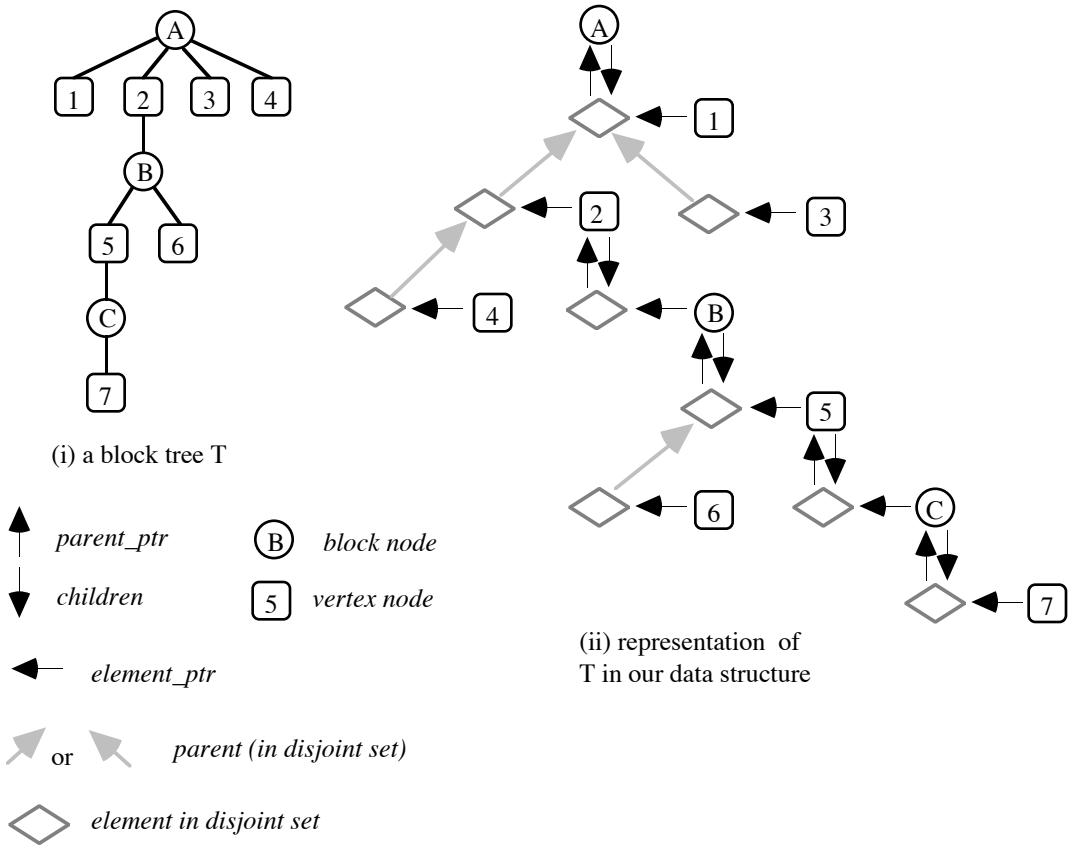



Figure 5. Block tree data structure in the first implementation
Each block node or vertex node may have two diamonds linked to it; the diamond directly under it is the root of the disjoint set that represents its children; the one on its immediate left represents itself in the children set of its parent (i.e. the disjoint set that represents its siblings).

data structure with *dynamic holes*. In Section 5, we present another mixed block tree data structure, which uses *static holes* instead of *dynamic holes* to deal with deletions. In Section 6, we describe a fourth implementation which uses mixed block tree, *static holes* and the standard disjoint set data structure. In each of these sections, we analyze the running time for the respective implementations. Section 7 concludes the discussion of biconnectivity with an experimental evaluation and analysis of all four of the implementations. The first of our four implementations is described in [3], and hence Section 3 is very brief.

### 3. Implementation I: Homogeneous Block Tree with Dynamic holes

In this implementation we make use of the modified disjoint set structure of Section 2.1.3 to maintain the children of the block tree nodes, since we have to deal with deletions in the disjoint sets. The details of this implementation are discussed in [3]. A minor difference in the current implementation with the one described in [3] is that the *node_ptr* field in the elements of the disjoint set structure is removed.

Children sets of nodes in a block tree are maintained using the disjoint set data structure, so that merging the children of two nodes can be done in one pointer step. 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 5). This implementation is labeled "homogeneous" since the children set of every node in a block tree is maintained by a disjoint set.

## 4. Implementation II: Mixed Block Tree with Dynamic holes

Our second implementation does not use disjoint sets to represent block nodes (see Figure 6). At the expense of uniformity throughout the data structure, it is possible to eliminate the disjoint set representation of block nodes and use direct pointers from block nodes to their parent vertex nodes. With this change, every traversal from a block node to a vertex node is reduced to constant time instead of one pointer step. Additionally, *dynamic holes* caused by *block_condense* are eliminated. We now describe this implementation in detail.

### 4.1 Maintaining Connected Components On-line

The data structure used to maintain connected components is exactly the disjoint set structure detailed in Section 2.1.1. The functions used to represent, update and answer queries of connected components have a one-to-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.

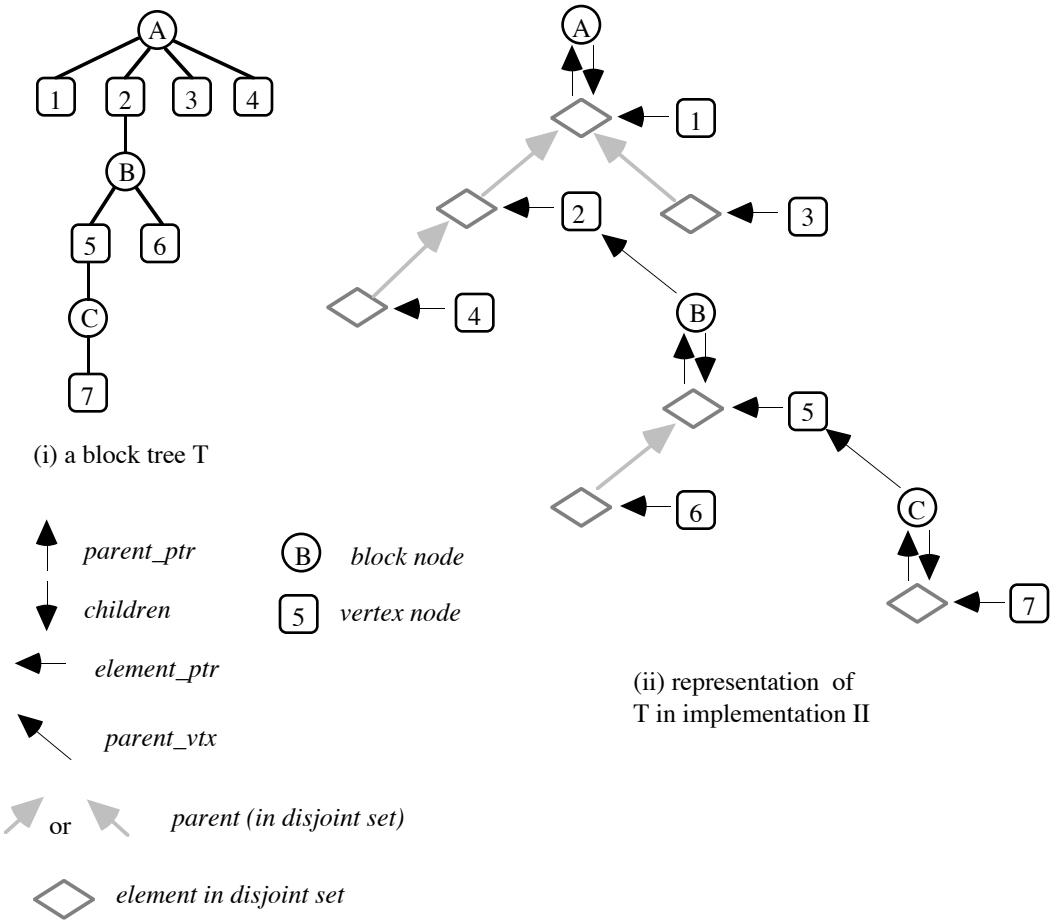



Figure 6. Block tree data structure in the second implementation

### 4.2 Organization of the Block Forest

The set of children vertex nodes of any block node is implemented using the disjoint set structure presented in Section 2.1.3. Each block node, with the exception of the root, has a direct pointer to its parent vertex node.

Each block node $B$ has the following fields:

- *label*: used to identify the block $B$
- *parent_vtx*: a pointer to the parent cut-vertex node of $B$, NULL $B$ is the root
- *children*: a disjoint set which contains the children vertices in $B$, some of these vertices may be cut vertices.
- *child_cnt*: the number of children in the children set, counts cut vertices twice.

Each vertex node $v$ has the following fields:

- *label*: used to identify the vertex node $v$
- *cc_ptr*: a pointer to the disjoint set which represents the connected component to which $v$ belongs.
- *element_ptr*: a pointer into the children set of $v$'s parent.

### 4.3 Functions for the Block Tree

The operations on a block tree 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 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 7). 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 $v$.

The details of the last four functions above are described in the following subsections.

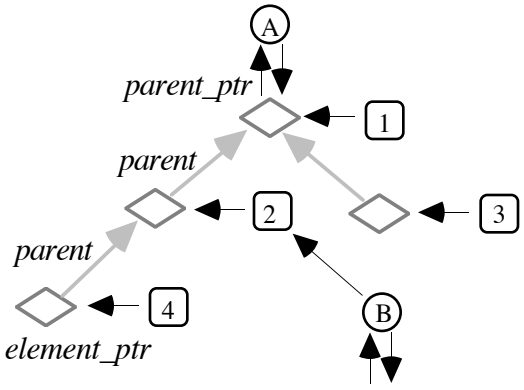



Figure 7. *block_parent*(4) = A

### 4.3.1 Finding Blocks

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

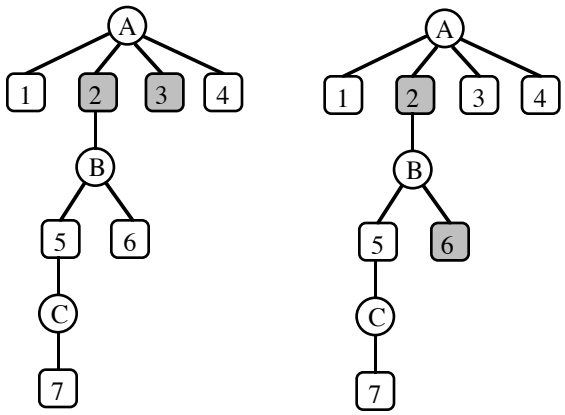
LEMMA 1. *Two vertices are in the same block iff they are siblings or one is the grandparent of the other in the block tree.*

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 8). []

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.



(i) siblings (ii) grandparent and grandchild

Figure 8. Vertices in the same block

### 4.3.2 Everting a Tree

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 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 9). To do so in our data structure, for every vertex node $u$ along the path from $x$ to the root, we must remove $u$ from *block_parent(u).children*, and then make *block_parent(u).parent_vtx* point to $u$. We remove $u$ from the disjoint set of its siblings by creating a *dynamic hole* that can be reused as described Section 2.1.3. This *dynamic hole* is later reused by $u$'s grandparent unless $u$ has no grandparent (see Figure 10).

Given a vertex 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 $x$:

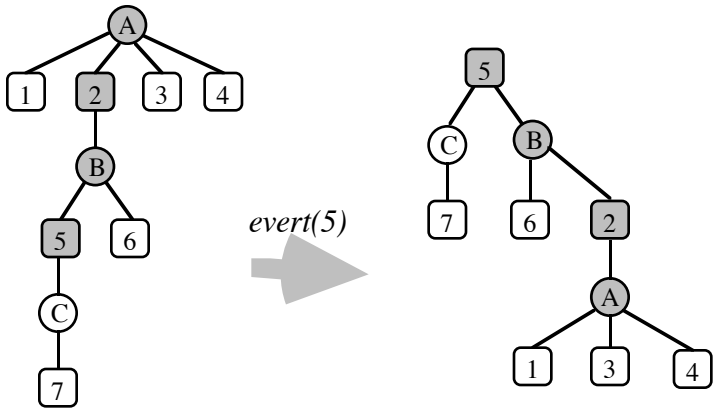



Figure 9. Eversion reverses the parent-child relationship along the path from the given vertex to the root.

```
procedure block_evert(x: vertex node)
    { initialize pointers to process first node }

    curr := x;
    parent := block_parent(curr);
    grandparent := block_parent(parent);

    { if the tree contains one vertex, eversion produces just a vertex node }
    if (parent→child_cnt = 1) && (grandparent = null)
        curr→element_ptr := null;
        free parent
        return;

    { sever the current node from its set and save its element as a hole to prime the loop }
    hole := curr→element_ptr;
    curr→element_ptr := null;

    while grandparent ≠ null
        { update the parent node to point to its child }
        parent→parent_vtx := curr;

        { update parent before severing grandparent }
        parent := block_parent(grandparent);

        { swap the current hole with the next node to be made a child }
        swap(&hole, &(grandparent→element_ptr));

        { update the other pointers }
        curr := grandparent;
```

```
        grandparent := parent→parent_vtx;

    end {while}

    { reverse the parent child relationship for the original root }
    parent→parent_vtx := curr;

    { the original root node will have one less child }
    parent→child_cnt := parent→child_cnt - 1;
end. { block_evert }
```

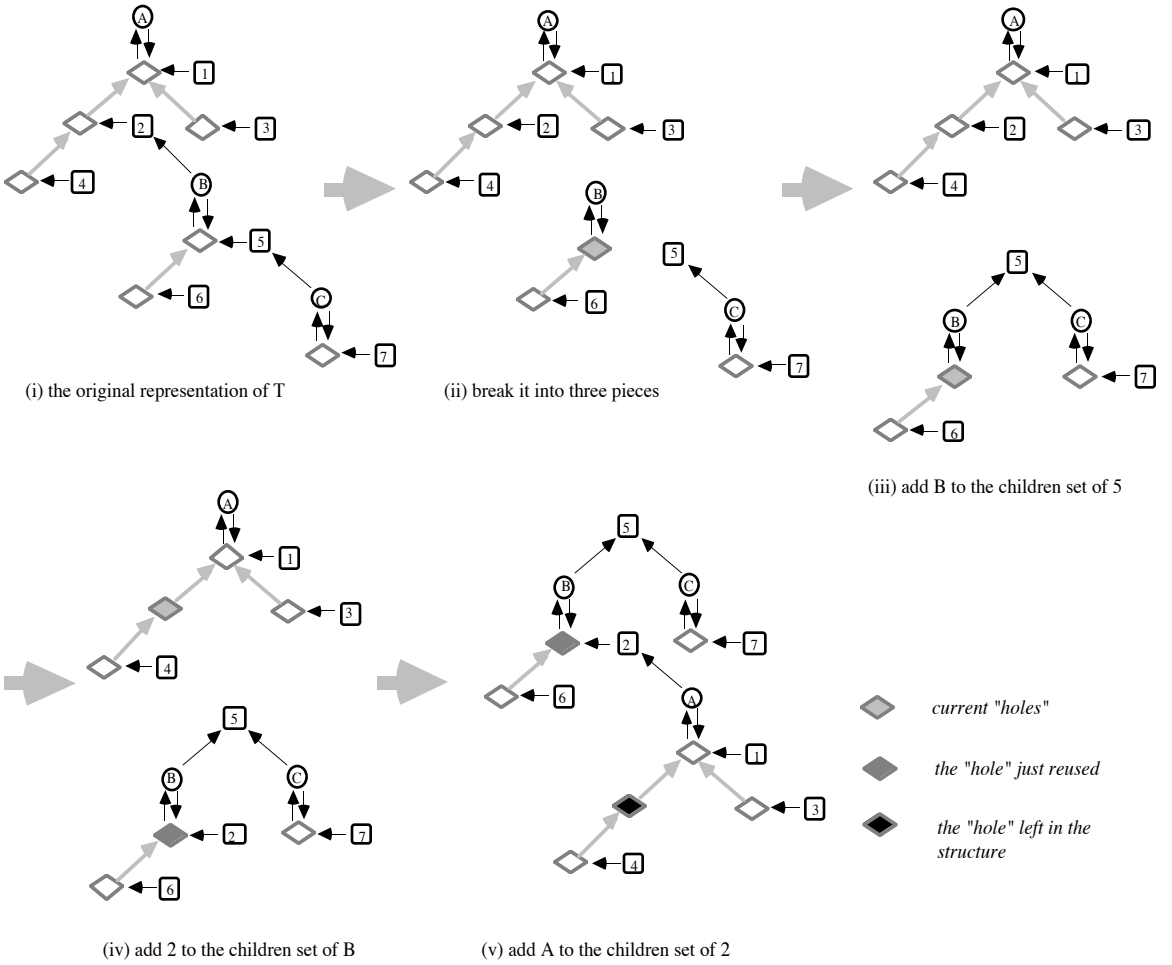



Figure 10. *block_evert*(5)

This algorithm will produce one dynamic hole (saved in the variable *hole*) per *block_evert* operation. It is important to recognize that the number of *dynamic 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 *dynamic 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 disjoint set data structure of Section 2.1.3.

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 [8] 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*. []

### 4.3.3 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 4.3.2, 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 temporarily the root of the smaller block tree. Also, 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)
    { if u and v belong to trees with only one vertex, then they can belong to the same block }
    if both u and v belong to trees with only one vertex then
        put v in  a new disjoint set, new_set;
        B := block_parent(u);
        (B→children) -> parent_node = null;
        B→children = ds_merge(u -> children, new_set);
        B→child_cnt := B→child_cnt + 1;

    B := block_new(v);
    B->parent_vtx := u;
end. { block_link }
```

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

### 4.3.4 Condensing a Path

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 11).

In order to condense these blocks, we first introduce two subroutines: *findlcav* and *condense_path*. The *findlcav* algorithm proceeds by walking up the tree simultaneously from its two vertex node arguments, until the paths intersect at their least common ancestor vertex node, which is returned by the function:

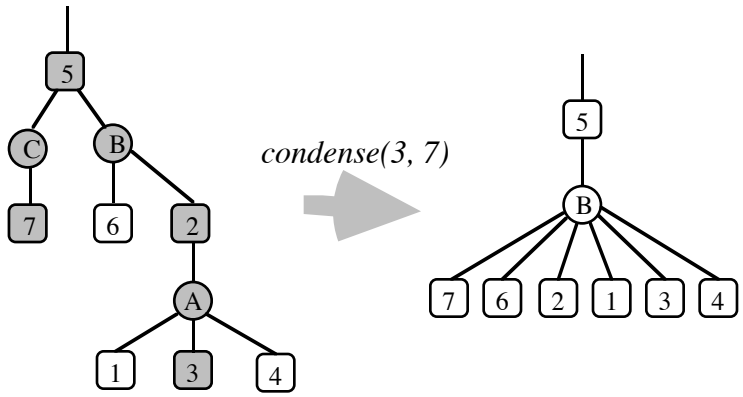



Figure 11. Condensation collapses all the blocks along the path between the two vertices into a single block.

***subroutine*** *findlcav(v1, v2: vertex node): vertex node*
{ find the least common ancestor vertex node of *v1* and *v2* by walking up the tree simultaneously from both of them, until the paths intersect at their lcav }
*lcav := v1;*
*save := v2;*
*while lcav.visited = false*
*lcav.visited := true;*
*if one of the paths has reached the root already then*
{ ignore the path that has reached the root from now on }
*lcav := block_parent(lcav)->parent_vtx;*
*if the current path reaches the root, get out of the while loop*
*else*
{ advance to the next node on the current path, and switch to the other path }
*tmp := block_parent(lcav)->parent_vtx;*
*lcav := save;*
*save := tmp;*
*unmark all the nodes that were marked;*
*return lcav;*
*end.* { *findlcav* }

The function *condense_path* takes two arguments, $x$ and $y$, where $x$ and $y$ are both vertex nodes and $y$ is an ancestor of $x$. All of the block nodes along the path from $x$ to $y$ 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.

***subroutine*** *condense_path(x: vertex node, y: vertex node): block node*
{ start with the parent block of $x$, initialize }
*b := block_parent(x);*
*v := b->parent_vtx;*
*u := empty set;*
{ process the blocks along the path one by one, until we reach the lcav if it is a block, or until its child if it is vertex }
*while v ≠ y*
*add the children set of b to u;*
*free(b);*
*b := block_parent(v);*
*v := b->parent_vtx;*
*add u into the children set of b;*
*return b;*
*end.* { *condense_path* }

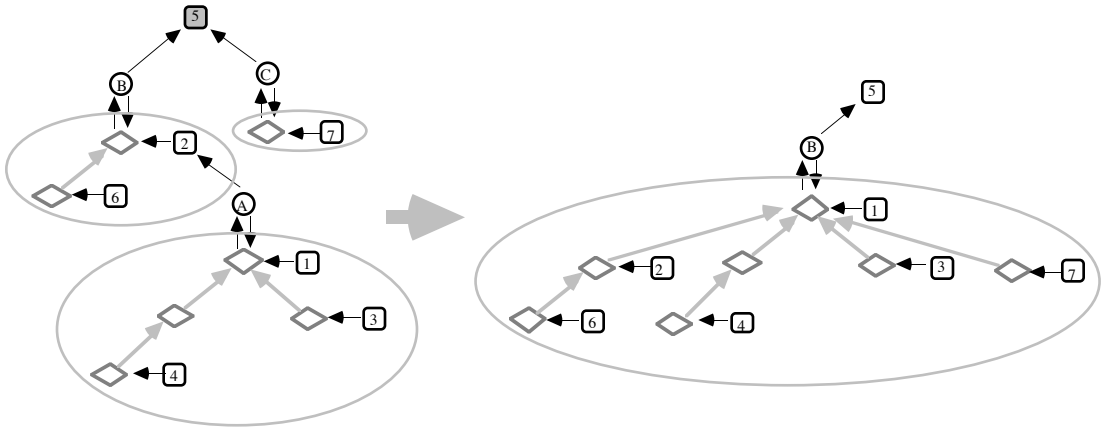



Figure 12. *block_condense*(3, 7)

Two vertex nodes that are in the same connected component and different blocks are given as the arguments to *block_condense*. Using *findlcav* and *condense_path*, the algorithm is the following (also see Figure 12):

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

Subroutine *findlcav* takes $O(P)$ pointer steps, where $P$ is the length of the path between the two given vertices (see *findpath*, [8]), 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.

### 4.4 Exported Functions to Maintain Connectivity and Biconnectivity

The data structures presented above maintain graph connectivity and biconnectivity properties separately. 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 prove the following lemma:

LEMMA 3. *In any sequence of insert_edge operations, there can be at most n-1 dynamic 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 *dynamic hole* in the children set of the root node, there can be at most $n-1$ such *dynamic holes* in the block forest. []

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 *dynamic holes* created by eversions. Hence, there will be at most $2n$-1 elements in all the disjoint sets in the block forest. This means the running time of $k$ *find* and *merge* operations will be $O(k\,\alpha(k, 2n\text{-}1))$, and the amortized time per operation will be $O(\alpha(k, 2n\text{-}1))$. The proof given below is similar to the one for implementation I [3].

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

PROOF. Westbrook and Tarjan [8] 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 $\Omega(n)$, it is shown in [8] that the total number of pointer steps required is $O(m + n\log n)$. However, this analysis does not take into account the *dynamic holes* that need to be created in the block tree data structure by *block_evert* .

By the analysis given in [8], 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 *dynamic holes*. From Lemma 3 it follows that $n' \leq 2n\text{-}1$, hence $n'$ is $O(n)$. Hence, our data structures can be maintained in $O(m + n\log n)$ time. []

**5. Implementation III: Mixed Block Tree with Static Holes**

In this implementation, we have no *dynamic holes*. Instead, we make use of *static holes*, which are created when newly created blocks have only cut vertices as their children. In Section 5.1, we describe the new block tree data structure, and in Section 5.2 we give descriptions of each of the functions needed to maintain the block forest. In Section 5.3 we prove the overall running time of this implementation.

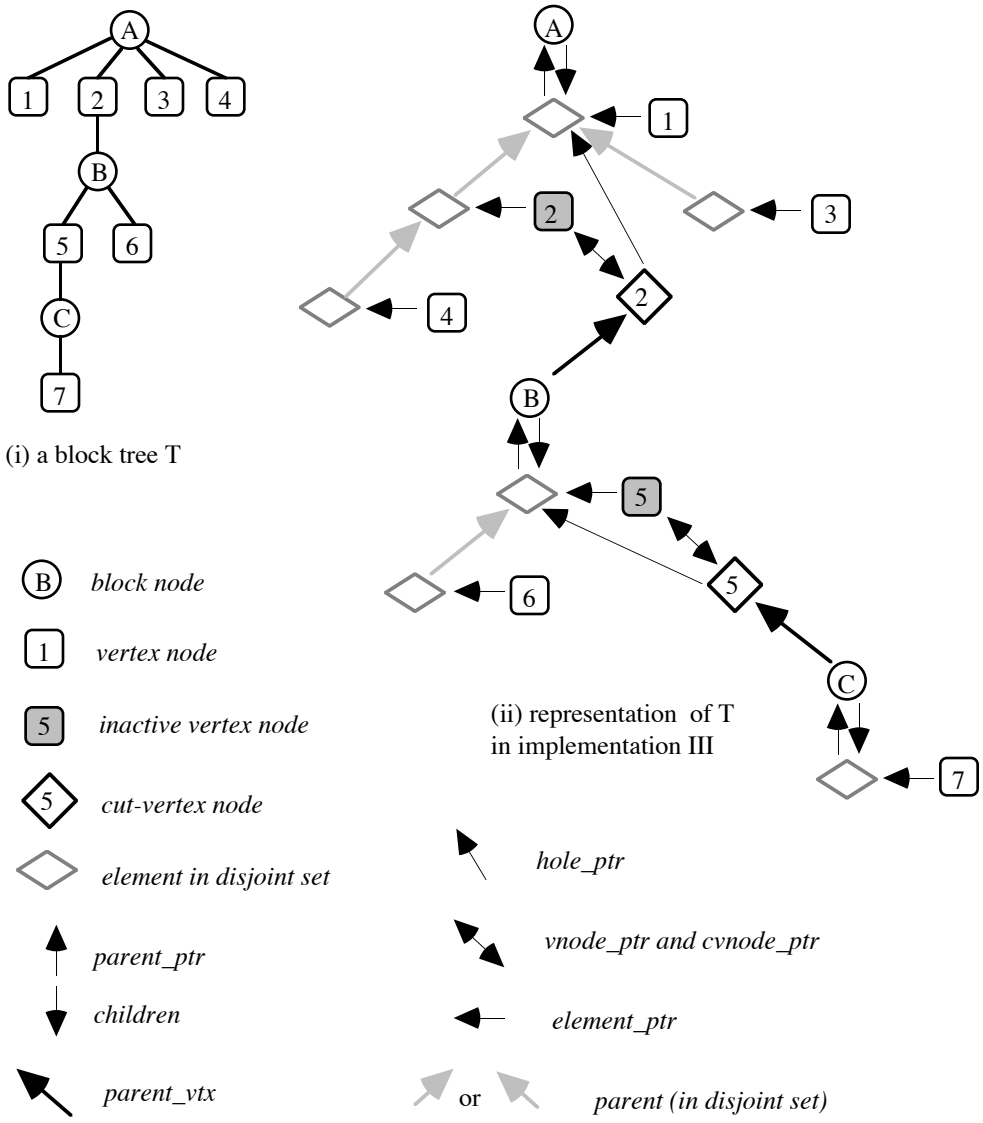



Figure 13. Block tree data structure in the third implementation

### 5.1 Data Structure

The data structure for the block tree in this implementation consists of block nodes, vertex nodes and cut-vertex nodes. The vertex nodes are organized into disjoint sets that represent children sets of different block nodes as in previous implementations. A cut-vertex node is a virtual copy of and doubly linked to its correspondent vertex node. A cut-vertex has a pointer to one of the elements in the children set of its parent block. This way its parent can be accessed while the merging of children sets of different blocks can be done efficiently. A cut-vertex node is created and deleted dynamically, while a vertex node is never deleted once it is created, instead it goes into an obsolete state when a cut-vertex node representing it exists. Block nodes, then, point to their parent cut-vertex nodes, not vertex nodes (see Figure 13).

Cut-vertex nodes can only be created when two distinct components are linked together. When these cut-vertex nodes are created, their corresponding vertex nodes become obsolete in the data structure. For example, if $u$ becomes a cut vertex in the graph, the corresponding vertex node in the block tree will point to the newly created cut-vertex node. From that point on, all operations which take $u$ as an argument will use the cut-vertex node instead of the vertex node for information about $u$, unless $u$ ceases to be a cut-vertex.

In the linking process, it is also possible that a new block $B$ needs to be created between the two cut-vertices, $u$ and $v$. $B$ will be the parent of one of these cut-vertices, say $u$, so $u$ must point to an element in the children set of $B$. Since $B$ has no children at this point, a *static hole* is created as a singleton element to represent the empty children set of $B$. Then, $v$ can be linked to this *static hole* to complete the linking process.

Each block node $B$ has the following fields:

- *label*: used to identify the block $B$
- *parent_vtx*: a pointer to the parent cut-vertex node of $B$, NULL $B$ is the root
- *children*: a disjoint set which contains the children vertices in $B$, some of these vertices may be cut vertices.
- *child_cnt*: the number of children in the children set, counts cut vertices twice.

Each vertex node $v$ has the following fields:

- *label*: used to identify the vertex node $v$
- *cc_ptr*: a pointer to the disjoint set which represents the connected component to which $v$ belongs.
- *element_ptr*: a pointer into the children set of $v$'s parent.
- *cvnode_ptr*: a pointer to a cut-vertex node iff $v$ is a cut-vertex, NULL otherwise.
- *block_cnt*: the number of blocks to which $v$ belongs, used to check if $v$ has a cut-vertex.

Each cut-vertex node $c$ has the following fields:

- *vnode_ptr*: a pointer back to the vertex node to which $c$ is attached
- *hole_ptr*: a pointer to a disjoint set element which gives $c$ membership in another block.
- *visited*: a boolean variable to indicate whether $v$ has been visited, used in *block_condense*

### 5.2 Functions for the Block Tree

The functions described below manipulate the block tree data structure in the same way as previous implementations. A minor difference in this implementation is that eversion and linking are done in the same function, *block_evert&link*, this change is inconsequential since an eversion is always followed by a link.

#### 5.2.1 block_new

The operation *block_new* allows us to create a new block node in the block tree and initialize that block with a vertex node $v$. However, *block_new* can also be called with a NULL argument, which creates an empty block node with one *static hole*. The need for this is shown in Section 5.2.4. This function is executed as follows:

```
procedure block_new(v: vertex node): block node
    { create a new singleton disjoint set for the new block }
    new_set := ds_new();

    { create the new block node }
    new_block → label := get_next_block_label();
    new_block → parent_vtx := NULL;
    new_block → child_cnt := 0;

    { attach the new set to the new block }
    new_block → children := new_set;
    new_set → parent_node := new_block;

    { attach v to the disjoint set if appropriate}
    if (v ≠ NULL) then
        attach vertex node v to new_set
        v → block_cnt := 1;
        new_block → child_cnt := 1;
```

```
    return(new_block)
end. {block_new}
```

The function *block_new* takes $O(1)$ time since all the operations it performs take constant time.

### 5.2.2 block_parent

The *block_parent* operation is used in the implementation to access the parent in the block tree of a vertex node $v$. If $v$ is a cut-vertex, then its corresponding cut-vertex node is used to access the parent block node. It executes as follows:

```
procedure block_parent(v: vertex node): block node
    { if v belongs to more than one block, it is a cut-vertex, so we access its cut-vertex node to find
      the block parent }
    if (v → block_cnt > 1)
        root := ds_find(v → cvnode_ptr → hole_ptr);
    { otherwise we access the sibling disjoint set of v for the block parent }
    else
        root := ds_find(v → element_ptr);

    return(root → parent_node);
end.
```

The function *block_parent* takes $O(1)$ pointer steps since *ds_find* is called only once.

### 5.2.3 block_find

The function *block_find* is used to check whether two vertex nodes, $u$, and $v$, are in the same biconnected component, i.e., belong to the same block in the block tree. We need this information in the exported function *insert_edge* so that we know how to modify the block tree upon an insertion of an edge to the graph.

First we check if $u$ or $v$ have the same block parent, if so, we simply return the common block node. Note that either $u$ or $v$ can be cut vertices yet still have the same parent. If $u$ and $v$ do not have the same parent, then one must be a cut vertex if they belong to a common block at all. So if either $u$ or $v$ is a cut vertex, we check if one the grandparent of the other, and if so, return the common block. Otherwise, we return NULL. The function is as follows:

```
procedure block_find(u, v: vertex node): block node
    { block_parent always returns the true parent of its argument, regardless of whether it is a cut
      vertex or not }
    parent_u := block_parent(u);
    parent_v := block_parent(v);

    { 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 (parent_u = parent_v) then
        return(parent_u);
    else if (u → block_cnt > 1) && (u → cvnode_ptr = parent_v → parent_vtx)
        return(parent_v);
    else if (v → block_cnt > 1) && (v → cvnode_ptr = parent_u → parent_vtx)
        return(parent_u);
    else
        return(NULL)
end. {block_find}
```

The function *block_find* takes $O(1)$ pointer steps since *ds_find* is called only twice.

### 5.2.4 block_evert&link

As input, *block_evert&link* takes the vertex nodes $s$ and $l$, where $s$ is the vertex node in the smaller block tree and $l$ is the vertex node in the larger tree. After the eversion and linking are done, *block_evert&link* returns a single block tree in which the vertices $s$ and $l$ are linked.

```
procedure block_evert&link(s, l: vertex node): block node
    { s is a vertex node in the smaller tree and l is a vertex node in the larger tree }
    parent_l := block_parent(l);
    parent_s := block_parent(s);
    grandparent_s := parent_s → parent_vtx;
    grandparent_l := parent_l → parent_vtx;

    { if l is the only vertex in the tree, then both trees are single vertex trees and we can merge their
      disjoint sets to create the new tree }
    if (parent_l → child_cnt = 1) && (grandparent_l = NULL)
        (parent_l → children) → parent_node := NULL;
        (parent_s → children) → parent_node := NULL;
        parent_l → children := ds_merge(parent_l → children, parent_s → children);
        (parent_l → children) → parent_node := parent_l;
        free(parent_s);
        return(parent_l);

    { if l does not already have a cut-vertex, then make a cut-vertex for it }
    if (l → block_cnt = 1)
        create a new cut-vertex node
        l → cvnode_ptr := new_cvnode;
        new_cvnode → vnode_ptr := l;
        new_cvnode → hole_ptr := parent_l → children;

    { if the smaller tree is a one vertex tree, then we need not evert the tree, it can be directly linked
      to the larger tree }
    if (parent_s → child_cnt = 1) && (grandparent_s = NULL)
        parent_s → parent_vtx := l → cvnode_ptr;
        l → block_cnt := l → block_cnt + 1;
        return(parent_s);

    { if s does not already have a cut-vertex, create a cut-vertex node for it }
    if (s → block_cnt = 1)
        create a new cut-vertex node
        s → cvnode_ptr := new_cvnode;
        new_cvnode → vnode_ptr := s;
        new_cvnode → hole_ptr := NULL;

    { if neither tree is a single vertex tree, we must evert the smaller tree and link it to the larger
      tree by creating a new block}
    curr := s → cvnode_ptr;

    { evert the tree from s to the root }
    while (grandparent_s ≠ NULL)
        { make parent_s a child of curr and save parent_s }
        parent_s → parent_vtx := curr;
        temp := parent_s;
```

```
        { update parent_s }
        parent_s := block_parent(grandparent_s → vnode_ptr);

        { reverse grandparent_s and old parent_s, i.e., temp }
        grandparent_s → hole_ptr := temp → children;
        curr := grandparent_s;
        grandparent_s := parent_s → parent_vtx;
    end {while}

    { original root loses one child due to eversion }
    parent → child_cnt := parent → child_cnt - 1;

    { relink the original root to the everted tree }
    parent_s → parent_vtx := curr;

    { link the everted tree to the larger tree by creating an empty block in between }
    new_block := block_new(NULL);
    (s → cvnode_ptr) → hole_ptr := new_block → children;
    new_block → parent_vtx := l → cvnode_ptr;
    new_block → child_cnt := 1;

    l → block_cnt := l → block_cnt + 1;

end. {block_evert}
```

The *block_evert&link* operation takes $O(P)$ pointer steps, where $P$ is the length of the path from $x$ to the root. All operations outside of the eversion loop take at most a constant number of pointer steps, hence we must show that the loop runs in $O(P)$ time. Each iteration of the loop takes a constant number of pointer steps then advances to the next node up the tree by modifying *curr*, *parent_s*, and *grandparent_s*. Then there are $P$ -1 iterations of the loop since *grandparent* will be *NULL* when *parent* is the original root of the tree. Hence *block_evert&link* takes $O(P)$ pointer steps.

### 5.2.5 block_condense

The function *block_condense* condenses all the nodes into a single block node along the path between its two argument vertex nodes.

The function *findlcav* is an auxiliary function which is used to find the least common ancestor vertex of two vertices, *v1* and *v2*. We define the least common ancestor vertex of *v1* and *v2* to be either the least common ancestor itself of *v1* and *v2* or the parent of the least common ancestor of *v1* and *v2*.

```
subroutine findlcav(v1, v2: vertex node): vertex node
    { find the least common ancestor vertex of v1 and v2  by walking up the tree simultaneously
      from both of them, until the paths intersect at their lcav vertex }
    lcav := v1→cvnode_ptr;
    save := v2→cvnode_ptr;
    while lcav.visited = false
        lcav.visited := true;
        if (save = NULL) then
            { ignore the path that has reached the root from now on }
            lcav := block_parent(lcav→vnode_ptr) → parent_vtx;
            if (lcav = NULL)
                break;
        else
            { advance to the next cut vertex node on the current path, and switch to the other path }
            tmp := block_parent(lcav→vnode_ptr) → parent_vtx;
            lcav := save;
```

```
            save := tmp;
        unmark all the nodes that were marked;
        return lcav → vnode_ptr;
end. { findlcav }
```

The other auxiliary function is *condense_path*, which takes two vertex nodes, *x* and one of its ancestors, *y*, and condenses the path from *x* to *y* into a single block whose children set contains all the vertices along that path, including *x* and *y*.

```
subroutine condense_path(x, y: vertex node): block node
    { start with the parent block of x, initialize }
    b := block_parent(x);
    {v is the current vertex node during the loop}
    if b is not the root
        v := (b → parent_vtx) → vnode_ptr;
    else
        v := NULL;
    u := empty set;

    { process the blocks along the path one by one, until we reach y }
    while v ≠ y
        add the children set of b to u;
        free(b);

        {update the variables for the next iteration}
        b := block_parent(v);
        if v ≠ NULL
            v → block_cnt := v → block_cnt - 1;
        if b is not the root
            v := (b → parent_vtx) → vnode_ptr;
        else
            v := NULL;
    end {while }

    add u into the children set of b;
    return b;
end. { condense_path }
```

Using *findlcav* and *condense_path*, *block_condense* is as follows:

```
procedure block_condense(v1, v2: vertex node)
    { find the least common ancestor vertex of v1 and v2 }
    lcav := findlcav(v1, v2);

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

The cost of executing block_condense is proportional to the cost of executing *find_lcav* and *condense_path* since *block_condense* calls these functions a constant number of times. The *find_lcav* subroutine takes at most $2P$ pointer steps, where $P$ is the length of the path from either of the vertices to the root. This is because each iteration advances up the tree, and can only advance as far as the root. The *condense_path* also takes at most $2P$ pointer steps since it merges sets from both vertices to the *lcav*. Hence, the total cost of *block_condense* is $O(P)$.

### 5.3 Running Time Analysis

There is one important difference between *dynamic holes* and *static holes*. The fact that *dynamic holes* are reused is crucial in establishing a bound for the total number of holes in the first two implementations, i.e., we reuse *dynamic holes* so that only one remains each time *block_evert* is called. In Implementation III, once we create a *static hole* in a disjoint set, it is never removed from its parent block. We need a different analysis to establish a bound on the number of *static holes*, as the following lemma shows:

LEMMA 5. *In any sequence of insert_edge operations, there can be at most n static holes created in the block tree data structure.*

PROOF. Eversion does not create *static holes*, since we simply reverse the parent pointers of block nodes and cut-vertex nodes without having to manipulate elements in disjoint sets. The *block_condense* operation only merges together disjoint sets, so it too does not create *static holes*. The only *static holes* created are in the *block_evert&link* operation when two trees are being linked together, since vertices which were not previously cut vertices may now become cut vertices. However, *block_evert&link* occurs at most $n$-1 times since there are at most $n$ block trees, i.e., distinct connected components, at any time. []

With this lemma, Theorem 1 in Section 4.4 still holds since total number of elements in the disjoint sets of the block tree including *static holes* is in $O(n)$ where $n$ is the total number of vertices in the graph. Hence the overall running time of this implementation is in $O(n \log n + m)$, where $m$ is number of updates and queries to the graph.

## 6. Implementation IV: A Mixed Block Tree with Static Holes and Standard Disjoint Set Data Structure

In this implementation, instead of using the disjoint set data structure that allows deletions in Section 2.1.3, we use the standard disjoint set data structure described in Section 2.1.1. Then, instead of using an abstract representation of the elements of a disjoint set, a vertex node itself is an element of the disjoint set (see Figure 14). This implementation is exactly the same as Implementation III, except for the use of the standard disjoint set data structure. This was not possible in Implementations I and II, since they used *dynamic holes* for deletions in the disjoint set data structure. Since Implementation IV uses the standard disjoint set structure, separate sets of functions are used to maintain the disjoint sets used for connectivity and in the block tree.

### 6.1 Data Structure

Each block node $B$ has the following fields:

- *label*: used to identify the block $B$
- *parent_vtx*: a pointer to the parent cut-vertex node of $B$, NULL $B$ is the root
- *children*: a disjoint set which contains the children vertices in $B$, some of these vertices may be cut vertices.
- *child_cnt*:: the number of children in the children set, counts cut vertices twice.

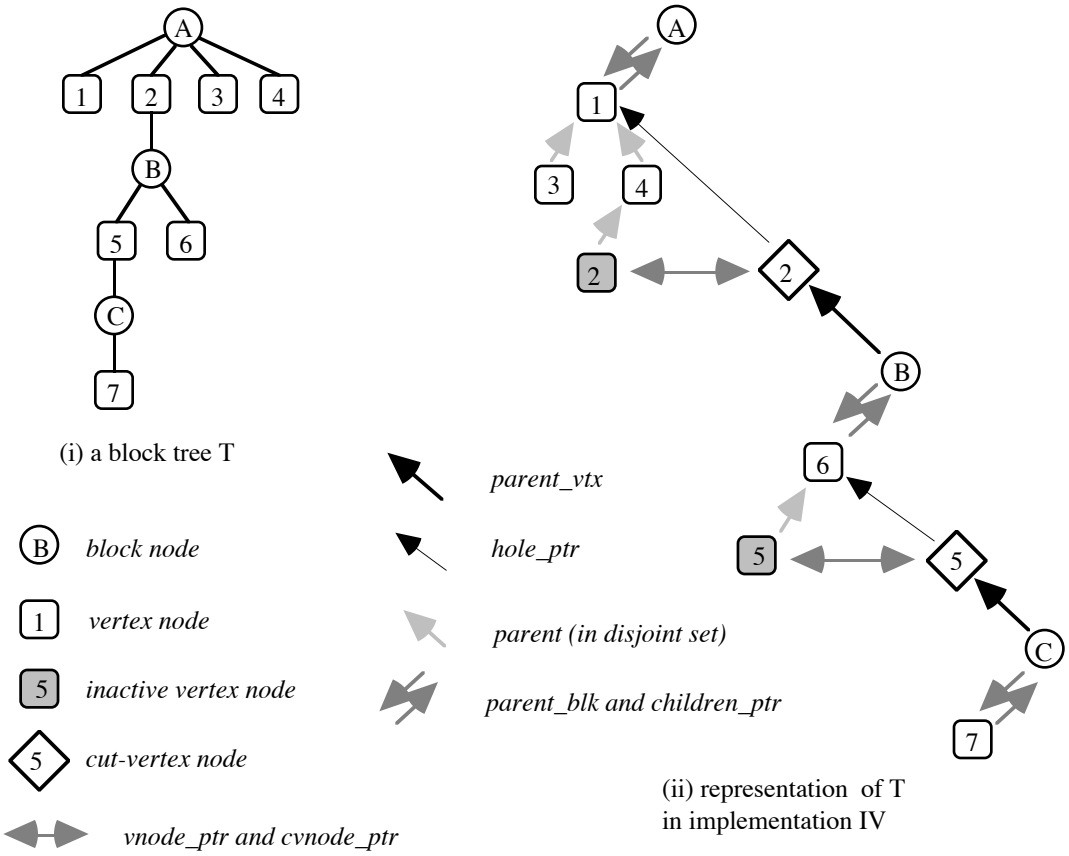



Figure 14. Block tree data structure in the fourth implementation

Each vertex node $v$ has the following fields:

- *label*: used to identify the vertex node $v$
- *cc_ptr*: a pointer to the disjoint set which represents the connected component to which $v$ belongs.
- *cc_size*: the size of the connected components that the current vertex represents.
- *bcc_ptr*: a pointer to the parent element of $v$ in the disjoint set representing its block.
- *bcc_size*: the size of the block in which $v$ belongs.
- *parent_blk:* a pointer to $v$'s parent block node.
- *cvnode_ptr*: a pointer to a cut-vertex node iff $v$ is a cut-vertex, NULL otherwise.
- *block_cnt*: the number of blocks to which $v$ belongs.

Each cut-vertex node $c$ has the following fields:

- *vnode_ptr*: a pointer back to the vertex node to which $c$ is attached
- *hole_ptr*: a pointer to a disjoint set element which gives $c$ membership in another block.
- *visited*: a boolean variable to indicate whether $v$ has been visited, used in *block_condense*

### 6.2 Functions to maintain the Connected Components

The standard disjoint set data structure of Section 2.1.1 is used, i.e., the vertex nodes are used as actual elements in the disjoint sets representing the connected components. As a result, separate disjoint set functions need to be implemented using the vertex nodes as the elements. The pseudo code below closely resembles the functions *ds_new*, *ds_find*, and *ds_merge*.

```
procedure cc_new_vertex(label): vertex node
    { create and initialize the new vertex node}
    v = new vertex node;
    v →label :=  label;
    vertices[v →label] := v;
    v →cc_ptr := v;
    v →cc_size := 1;
    v →bcc_ptr :=v;
    v →bcc_size := 1;
    v →parent_blk := NULL;
    v →cvnode_ptr := NULL;
    v →block_cnt := 0;
    return v;

procedure cc_find(v: vertex node): vertex node
    { find the root of the set }
    root := v;
    while (root →cc_ptr != root)
        root := root  →cc_ptr;
    tmp := v;

    { path compression }
    while (tmp != root)
        parent := tmp  →cc_ptr;
        tmp →cc_ptr := root;
        tmp := parent;

    { return the root of the set as its label }
    return(root);

procedure cc_merge(u, v: vertex node): vertex node
    { check for empty sets }
    if u = NULL
        return v;
    if v = NULL;
        return u;

    { make the smaller connected components a child branch of the larger one }
    if u →cc_size >= v →cc_size
        v →cc_ptr = u;
        u →cc_size = u →cc_size + v →cc_size;
        return u;
    else
        u →cc_ptr = v;
        v →cc_size = v →cc_size + u →cc_size;
        return v;
```

### 6.3 Functions to maintain the disjoint set of vertices in a block

As in the previous section, separate disjoint set functions are needed to maintain children sets of the block nodes in the block forest:

```
procedure bcc_find(v: vertex node): vertex node
    { find the root of the set }
    root := v;
    while (root →bcc_ptr != root)
        root := root  →bcc_ptr;
```

```
    tmp := v;

    { path compression }
    while (tmp != root)
        parent := tmp →bcc_ptr;
        tmp →bcc_ptr := root;
        tmp := parent;

    { return the root of the set as its label }
    return(root);

procedure bcc_merge(u, v: vertex node): vertex node
    { check for empty sets }
    if u = NULL
        return v;
    if v = NULL;
        return u;

    { make the smaller connected components a child branch of the larger one }
    if u →bcc_size >= v →bcc_size
        v →bcc_ptr = u;
        u →bcc_size = u →bcc_size + v →bcc_size;
        return u;
    else
        u →bcc_ptr = v;
        v →bcc_size = v →bcc_size + u →bcc_size;
        return v;
```

### 6.4 Functions for the Block Tree

In this implementation, the only block tree functions that changed due the use of the standard disjoint set structure are *block_new* and *block_parent*. This is because the disjoint set functions are only invoked by these functions, and all other functions access the disjoint sets through *block_new* and *block_parent*.

```
procedure block_new(v: vertex node): block node
    { create the new block node }
    new_block → label := get_next_block_label();
    new_block → parent_vtx := NULL;
    new_block → child_cnt := 0;

    { create a static hole vertex node if requested }
    if (v = NULL) then
        v = cc_new_vertex(0);

    v →parent_blk := new_block;
    v →block_cnt := 1;

    new_block → children := v;
    new_block → child_cnt := 1;

    return(new_block)
end. {block_new}

procedure block_parent(v: vertex node): block node
    { if v belongs to more than one block, it is a cut-vertex, so we access its cut-vertex node to find
      the block parent }
```

| # vertices | # edges | # updts. | init. time | std dev | updt time | std dev | time/op |
|---|---|---|---|---|---|---|---|
| 500 | 250 | 50 | 0.020464 | 0.000971 | 0.001564 | 0.000315 | 0.000031 |
| 500 | 250 | 500 | 0.020272 | 0.000293 | 0.008950 | 0.000198 | 0.000018 |
| 500 | 250 | 2000 | 0.020719 | 0.001827 | 0.016133 | 0.000544 | 0.000008 |
| 500 | 500 | 50 | 0.028000 | 0.000420 | 0.000968 | 0.000199 | 0.000019 |
| 500 | 500 | 500 | 0.027689 | 0.000250 | 0.004154 | 0.000348 | 0.000008 |
| 500 | 500 | 2000 | 0.027728 | 0.000279 | 0.008876 | 0.000488 | 0.000004 |
| 1000 | 500 | 100 | 0.039076 | 0.000356 | 0.002244 | 0.000170 | 0.000022 |
| 1000 | 500 | 1000 | 0.038981 | 0.000691 | 0.017091 | 0.000618 | 0.000017 |
| 1000 | 500 | 5000 | 0.039031 | 0.000525 | 0.035871 | 0.000513 | 0.000007 |
| 1000 | 1000 | 100 | 0.054043 | 0.000475 | 0.001803 | 0.000249 | 0.000018 |
| 1000 | 1000 | 1000 | 0.053866 | 0.000439 | 0.009110 | 0.000266 | 0.000009 |
| 1000 | 1000 | 5000 | 0.053799 | 0.000307 | 0.021395 | 0.000326 | 0.000004 |
| 5000 | 2500 | 500 | 0.192875 | 0.001209 | 0.010901 | 0.000601 | 0.000022 |
| 5000 | 2500 | 5000 | 0.192640 | 0.000652 | 0.090828 | 0.001638 | 0.000018 |
| 5000 | 2500 | 30000 | 0.192216 | 0.000802 | 0.220937 | 0.002157 | 0.000007 |
| 5000 | 5000 | 500 | 0.273097 | 0.001267 | 0.006852 | 0.000396 | 0.000014 |
| 5000 | 5000 | 5000 | 0.272913 | 0.001392 | 0.048924 | 0.000757 | 0.000010 |
| 5000 | 5000 | 30000 | 0.273021 | 0.002219 | 0.145002 | 0.000960 | 0.000005 |
| 10000 | 5000 | 1000 | 0.391531 | 0.009111 | 0.023026 | 0.000671 | 0.000023 |
| 10000 | 5000 | 10000 | 0.387831 | 0.000783 | 0.186746 | 0.002713 | 0.000019 |
| 10000 | 5000 | 70000 | 0.387806 | 0.000580 | 0.498349 | 0.007722 | 0.000007 |
| 10000 | 10000 | 1000 | 0.548439 | 0.002044 | 0.014015 | 0.000501 | 0.000014 |
| 10000 | 10000 | 10000 | 0.548296 | 0.002217 | 0.101678 | 0.001047 | 0.000010 |
| 10000 | 10000 | 70000 | 0.547809 | 0.002050 | 0.340697 | 0.002272 | 0.000005 |
| 20000 | 10000 | 2000 | 0.790035 | 0.003252 | 0.044760 | 0.000798 | 0.000022 |
| 20000 | 10000 | 20000 | 0.783890 | 0.001780 | 0.364341 | 0.007040 | 0.000018 |
| 20000 | 10000 | 150000 | 0.782605 | 0.001146 | 1.064185 | 0.009208 | 0.000007 |
| 20000 | 20000 | 2000 | 1.094907 | 0.003345 | 0.025839 | 0.000611 | 0.000013 |
| 20000 | 20000 | 20000 | 1.094131 | 0.003331 | 0.203280 | 0.002425 | 0.000010 |
| 20000 | 20000 | 150000 | 1.093537 | 0.003425 | 0.770003 | 0.007984 | 0.000005 |

Table 1. Statistics for test data run with first implementation

```
    if (v → block_cnt > 1)
        root := bcc_find(v → cvnode_ptr → hole_ptr);
    { otherwise we access the sibling disjoint set of v for the block parent }
    else
        root := bcc_find(v → bcc_ptr);

    return(root → parent_blk);
end.
```

We note that the running time for this implementation is the same as for Implementation III to within a constant factor. One would expect Implementation IV to be faster than III since the former uses the standard disjoint set structure; however III has a more elegant representation.

## 7. Experimental Results and Analysis

To evaluate the performance of the algorithms above, we ran all four implementations described above with a variety of initial graphs and update/query sequences. In Section 7.1, we describe the test environment and the test data that we used; in Section 7.2, we show the timing results for all four implementations. Finally in Section 7.3, we analyze the timing results and make relative comparisons among them.

| # vertices | # edges | # updts. | init. time | std dev | updt time | std dev | time/op. |
|---|---|---|---|---|---|---|---|
| 500 | 250 | 50 | 0.016998 | 0.001288 | 0.000979 | 0.000056 | 0.000020 |
| 500 | 250 | 500 | 0.016363 | 0.000262 | 0.005697 | 0.000091 | 0.000011 |
| 500 | 250 | 2000 | 0.016993 | 0.001585 | 0.011722 | 0.000106 | 0.000006 |
| 500 | 500 | 50 | 0.020954 | 0.000151 | 0.000825 | 0.000080 | 0.000017 |
| 500 | 500 | 500 | 0.021030 | 0.000303 | 0.003498 | 0.000165 | 0.000007 |
| 500 | 500 | 2000 | 0.020948 | 0.000185 | 0.007481 | 0.000114 | 0.000004 |
| 1000 | 500 | 100 | 0.031576 | 0.000140 | 0.001520 | 0.000080 | 0.000015 |
| 1000 | 500 | 1000 | 0.031432 | 0.000148 | 0.011349 | 0.000446 | 0.000011 |
| 1000 | 500 | 5000 | 0.031463 | 0.000179 | 0.025890 | 0.000299 | 0.000005 |
| 1000 | 1000 | 100 | 0.041012 | 0.000362 | 0.001257 | 0.000087 | 0.000013 |
| 1000 | 1000 | 1000 | 0.040741 | 0.000197 | 0.006737 | 0.000224 | 0.000007 |
| 1000 | 1000 | 5000 | 0.040774 | 0.000272 | 0.017208 | 0.000206 | 0.000003 |
| 5000 | 2500 | 500 | 0.154736 | 0.000856 | 0.007270 | 0.000415 | 0.000015 |
| 5000 | 2500 | 5000 | 0.155924 | 0.004701 | 0.060193 | 0.000668 | 0.000012 |
| 5000 | 2500 | 30000 | 0.154294 | 0.000353 | 0.164520 | 0.002684 | 0.000005 |
| 5000 | 5000 | 500 | 0.206186 | 0.000688 | 0.004985 | 0.000200 | 0.000010 |
| 5000 | 5000 | 5000 | 0.206220 | 0.000433 | 0.035827 | 0.000464 | 0.000007 |
| 5000 | 5000 | 30000 | 0.206190 | 0.000442 | 0.117246 | 0.000672 | 0.000004 |
| 10000 | 5000 | 1000 | 0.313044 | 0.000584 | 0.014899 | 0.000496 | 0.000015 |
| 10000 | 5000 | 10000 | 0.310812 | 0.000694 | 0.122622 | 0.000787 | 0.000012 |
| 10000 | 5000 | 70000 | 0.311011 | 0.000516 | 0.373678 | 0.001498 | 0.000005 |
| 10000 | 10000 | 1000 | 0.414716 | 0.000971 | 0.009844 | 0.000169 | 0.000010 |
| 10000 | 10000 | 10000 | 0.414532 | 0.001922 | 0.074317 | 0.000552 | 0.000007 |
| 10000 | 10000 | 70000 | 0.414045 | 0.000900 | 0.279543 | 0.000624 | 0.000004 |
| 20000 | 10000 | 2000 | 0.631166 | 0.002251 | 0.029004 | 0.000594 | 0.000015 |
| 20000 | 10000 | 20000 | 0.626562 | 0.007333 | 0.242678 | 0.001349 | 0.000012 |
| 20000 | 10000 | 150000 | 0.624115 | 0.000466 | 0.815021 | 0.005805 | 0.000005 |
| 20000 | 20000 | 2000 | 0.828986 | 0.003636 | 0.018631 | 0.000606 | 0.000009 |
| 20000 | 20000 | 20000 | 0.826884 | 0.001654 | 0.149229 | 0.001146 | 0.000007 |
| 20000 | 20000 | 150000 | 0.827888 | 0.002935 | 0.632414 | 0.002091 | 0.000004 |

Table 2. Statistics for test data run with second implementation

### 7.1 Test Environment and Test Data

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 each implementation 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% *insert_edge* operations and 50% *find_block* operations. 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 (and in fact, contain a Hamiltonian cycle, -- i.e. become biconnected) when the number of its edges reaches $n\ln n/2+o(n\ln n)$ [5], after which every operation will be within one block. At this stage the performance of the algorithm speeds up substantially from $\theta(\log n)$ amortized time per step to $\theta(\alpha(m, n))$ amortized time. Hence we take the length of the update/query sequence to be $n\ln n/2$, 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.

| # vertices | # edges | # updts. | init. time | std dev | updt time | std dev | time/op. |
|---|---|---|---|---|---|---|---|
| 500 | 250 | 50 | 0.013578 | 0.000132 | 0.000802 | 0.000065 | 0.000016 |
| 500 | 250 | 500 | 0.013564 | 0.000117 | 0.005277 | 0.000509 | 0.000011 |
| 500 | 250 | 2000 | 0.013591 | 0.000110 | 0.011365 | 0.000403 | 0.000006 |
| 500 | 500 | 50 | 0.017987 | 0.000338 | 0.000814 | 0.000080 | 0.000016 |
| 500 | 500 | 500 | 0.018070 | 0.000389 | 0.003435 | 0.000188 | 0.000007 |
| 500 | 500 | 2000 | 0.018002 | 0.000343 | 0.007703 | 0.000247 | 0.000004 |
| 1000 | 500 | 100 | 0.026620 | 0.000225 | 0.001177 | 0.000082 | 0.000012 |
| 1000 | 500 | 1000 | 0.026460 | 0.000114 | 0.010139 | 0.000516 | 0.000010 |
| 1000 | 500 | 5000 | 0.026591 | 0.000180 | 0.025115 | 0.000361 | 0.000005 |
| 1000 | 1000 | 100 | 0.034901 | 0.000557 | 0.001194 | 0.000108 | 0.000012 |
| 1000 | 1000 | 1000 | 0.034710 | 0.000508 | 0.006615 | 0.000249 | 0.000007 |
| 1000 | 1000 | 5000 | 0.034575 | 0.000497 | 0.017706 | 0.000236 | 0.000004 |
| 5000 | 2500 | 500 | 0.132637 | 0.000950 | 0.006164 | 0.000543 | 0.000012 |
| 5000 | 2500 | 5000 | 0.133249 | 0.003879 | 0.056099 | 0.000915 | 0.000011 |
| 5000 | 2500 | 30000 | 0.132014 | 0.000680 | 0.167698 | 0.001471 | 0.000006 |
| 5000 | 5000 | 500 | 0.179285 | 0.001052 | 0.004927 | 0.000270 | 0.000010 |
| 5000 | 5000 | 5000 | 0.179111 | 0.000998 | 0.036958 | 0.001320 | 0.000007 |
| 5000 | 5000 | 30000 | 0.179918 | 0.001705 | 0.127410 | 0.003708 | 0.000004 |
| 10000 | 5000 | 1000 | 0.269621 | 0.001014 | 0.013017 | 0.000582 | 0.000013 |
| 10000 | 5000 | 10000 | 0.266987 | 0.001407 | 0.116186 | 0.002028 | 0.000012 |
| 10000 | 5000 | 70000 | 0.268687 | 0.004644 | 0.395005 | 0.014894 | 0.000006 |
| 10000 | 10000 | 1000 | 0.362262 | 0.000762 | 0.009963 | 0.000326 | 0.000010 |
| 10000 | 10000 | 10000 | 0.360480 | 0.000671 | 0.077466 | 0.000772 | 0.000008 |
| 10000 | 10000 | 70000 | 0.360594 | 0.000537 | 0.307112 | 0.000796 | 0.000004 |
| 20000 | 10000 | 2000 | 0.546732 | 0.010076 | 0.027107 | 0.002289 | 0.000014 |
| 20000 | 10000 | 20000 | 0.541567 | 0.008510 | 0.229329 | 0.003906 | 0.000011 |
| 20000 | 10000 | 150000 | 0.539566 | 0.002051 | 0.842906 | 0.005191 | 0.000006 |
| 20000 | 20000 | 2000 | 0.726126 | 0.001899 | 0.018541 | 0.000450 | 0.000009 |
| 20000 | 20000 | 20000 | 0.724298 | 0.001673 | 0.152558 | 0.001412 | 0.000008 |
| 20000 | 20000 | 150000 | 0.724630 | 0.001912 | 0.683555 | 0.002264 | 0.000005 |

Table 3. Statistics for test data run with third implementation

### 7.2 Test Results

The times (given in seconds) obtained from our tests are listed in Tables 1-4 in ascending order of vertices and edges. A data point in the table consists of a number of vertices in the graph, an initial number of edges, and an update/query sequence size. For each data point, we give an initialization time and an update time, along with their respective standard deviations. We also give the time per operation for each update/query sequence.

### 7.3 Comparison and Analysis

In the course of our experiments, we examined if there was a clear difference in implementations with *dynamic holes* versus *static holes*. We knew that there would be fewer *static holes* than *dynamic holes*, but were not sure if the overhead of maintaining *static holes* would lead to slower performance. Similarly, we knew that mixed block tree data structures should be faster than homogeneous ones due to direct pointers, but did not know if maintaining certain nodes in the block tree differently would be too expensive.

After running all of the implementations, we plotted the results by initial graph density and update/query sequence size (see below). The results show that Implementation IV has the fastest performance and Implementation I is the slowest (see Table 1-4). From the plots, we found that for smaller update/query sequence sizes, the use of *static holes* with the modified disjoint set structure (Implementation III) led to slightly faster performance than Implementation II. However, for larger update/query sequence sizes, the overhead of maintaining both static holes and the modified disjoint sets structure caused Implementation II to overtake Implementation III. As noted above,

| # vertices | # edges | # updts. | init time | std dev | updt time | std dev | time/op. |
|---|---|---|---|---|---|---|---|
| 500 | 250 | 50 | 0.010421 | 0.000963 | 0.000763 | 0.000069 | 0.000015 |
| 500 | 250 | 500 | 0.009943 | 0.000104 | 0.005125 | 0.000398 | 0.000010 |
| 500 | 250 | 2000 | 0.009887 | 0.000063 | 0.010624 | 0.000266 | 0.000005 |
| 500 | 500 | 50 | 0.014139 | 0.000211 | 0.000780 | 0.000065 | 0.000016 |
| 500 | 500 | 500 | 0.014256 | 0.000445 | 0.003096 | 0.000124 | 0.000006 |
| 500 | 500 | 2000 | 0.014150 | 0.000245 | 0.006925 | 0.000190 | 0.000003 |
| 1000 | 500 | 100 | 0.018086 | 0.000283 | 0.001995 | 0.000314 | 0.000020 |
| 1000 | 500 | 1000 | 0.018178 | 0.000320 | 0.009936 | 0.000402 | 0.000010 |
| 1000 | 500 | 5000 | 0.018324 | 0.000863 | 0.023959 | 0.001627 | 0.000005 |
| 1000 | 1000 | 100 | 0.026632 | 0.000555 | 0.001212 | 0.000193 | 0.000012 |
| 1000 | 1000 | 1000 | 0.026218 | 0.000295 | 0.006112 | 0.000454 | 0.000006 |
| 1000 | 1000 | 5000 | 0.026372 | 0.000552 | 0.016491 | 0.001790 | 0.000003 |
| 5000 | 2500 | 500 | 0.089638 | 0.001414 | 0.005675 | 0.000289 | 0.000011 |
| 5000 | 2500 | 5000 | 0.088896 | 0.000327 | 0.052272 | 0.000981 | 0.000010 |
| 5000 | 2500 | 30000 | 0.089720 | 0.001942 | 0.146142 | 0.001224 | 0.000005 |
| 5000 | 5000 | 500 | 0.134579 | 0.000701 | 0.004271 | 0.000201 | 0.000009 |
| 5000 | 5000 | 5000 | 0.133818 | 0.000544 | 0.032419 | 0.000461 | 0.000006 |
| 5000 | 5000 | 30000 | 0.133976 | 0.000567 | 0.105393 | 0.000609 | 0.000004 |
| 10000 | 5000 | 1000 | 0.181467 | 0.000858 | 0.012411 | 0.000806 | 0.000012 |
| 10000 | 5000 | 10000 | 0.179751 | 0.000733 | 0.108466 | 0.001768 | 0.000011 |
| 10000 | 5000 | 70000 | 0.179690 | 0.000819 | 0.339014 | 0.001950 | 0.000005 |
| 10000 | 10000 | 1000 | 0.271756 | 0.001902 | 0.008997 | 0.000464 | 0.000009 |
| 10000 | 10000 | 10000 | 0.270138 | 0.001372 | 0.068564 | 0.000962 | 0.000007 |
| 10000 | 10000 | 70000 | 0.270128 | 0.000890 | 0.257652 | 0.004300 | 0.000004 |
| 20000 | 10000 | 2000 | 0.366658 | 0.001150 | 0.024595 | 0.000615 | 0.000012 |
| 20000 | 10000 | 20000 | 0.363703 | 0.001468 | 0.216970 | 0.004684 | 0.000011 |
| 20000 | 10000 | 150000 | 0.368074 | 0.014541 | 0.733604 | 0.003733 | 0.000005 |
| 20000 | 20000 | 2000 | 0.544021 | 0.002629 | 0.017049 | 0.000381 | 0.000009 |
| 20000 | 20000 | 20000 | 0.541997 | 0.002191 | 0.135053 | 0.001765 | 0.000007 |
| 20000 | 20000 | 150000 | 0.542366 | 0.002044 | 0.575943 | 0.003541 | 0.000004 |

Table 4. Statistics for test data run with fourth implementation

the use of *static holes* with the standard disjoint set structure in Implementation IV led to the best performance. However, Implementation II presents a better abstraction of data structures than Implementation IV, since its representation and maintenance of connectivity and biconnectivity is clearly separate.

We also noticed that in Implementation IV, the initialization time improved much more significantly than the update time. With profiling information obtained from the software Program Quantify 2.1 [4], we observed that this discrepancy is explained by the differences in the cost of *new_vertex* operations across implementations. The use of the standard disjoint set structure in Implementation IV meant that within each *new_vertex* call, a single memory allocation is made for the vertex node. In the other implementations, each *new_vertex* call makes three memory allocations, one for the vertex node itself, and two for its disjoint set representations. Since *new_vertex* operation is only invoked during the initialization, this explanation shows why the initialization time was improved much more than the update time in Implementation IV.

We also made a comparison between our implementations and those of Alberts et al. [1], the only other experimental study of dynamic graph algorithms that we knew of (see Table 5). Our implementations run significantly faster as 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. Alberts et al. maintain only connectivity, while our implementation maintains both connectivity and biconnectivity. However, the algorithms we implemented were incremental, whereas the algorithms tested by Alberts et al., are fully dynamic, and therefore handle edge deletions, which are harder to deal with. Hence it is not surprising that our algorithm runs much faster than the ones tested by Alberts et al.

| vertices | edges | #updt | Alberts | Imp. IV |
|---|---|---|---|---|
| 500 | 250 | 500 | 0.276* | 0.005125 |
| 500 | 500 | 500 | 0.3588* | 0.003096 |

Table 5. Common data points between Alberts et. al and Implementation IV
(* normalized time [2])

**Plots of Test Results**

The plots on the following pages were made from the test data given in Tables 1-4; all graphs have $n$ vertices. On each page, we compare Implementations I-IV for one update/query sequence size ($n/10$, $n$, $n\ln n - 2n$) on very sparse and sparse initial graphs.

## 8. Conclusion

In this paper, we have described four implementations of an algorithm for the maintenance of incremental biconnectivity. The algorithm allows insertions of edges and vertices and answers biconnectivity queries, i.e., "given two vertices of a graph, are they in the same biconnected component ?" In the course of implementing the algorithm given by [8], we discovered an issue that was not addressed in [8]: the need for deletions in disjoint sets to preserve the efficiency of the algorithm. We modified the standard disjoint set data structure and developed two new concepts in dealing with deletions: *dynamic holes* and *static holes*. We have implemented two versions of the algorithm which use *dynamic holes* and two which use *static holes*. With both *static* and *dynamic holes*, we prove that the performance of the overall block tree data structure with our disjoint set structure is still $O(n\log n + m)$, where $n$ is the number of vertices in the graph and $m$ is the total number of queries and updates. We tested the four implementations extensively and presented the experimental results and analysis.

***Acknowledgment.*** We would like to thank Madhukar Korupolu for extensive discussions in the early stages of this project. This paper is part of an undergraduate honors thesis project undertaken by Ramgopal Mettu and Yuke Zhao.

## References

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

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

[3] M. Korupolu, R. Mettu, V. Ramachandran, and Y. Zhao. Experimental Evaluation of Algorithms for Incremental Graph Connectivity and Biconnectivity. Presented at the *Fifth DIMACS Implementation Challenge*, 1996.

[4] Quantify 2.1. Version 2.1, Pure Corporation, 1996.

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

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

[7] 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.

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