Finding 3-shredders efficiently

Rajneesh Hegde*

July 2, 2002

Abstract

A shredder in an undirected graph is a set of vertices whose removal results in at least three components. A 3-shredder is a shredder of size three. We present an algorithm that, given a 3-connected graph, finds its 3-shredders in time proportional to the number of vertices and edges, when implemented on a RAM (random access machine).

1 Introduction

Connectivity is an important invariant of graphs. Efficient algorithms for determining the connectivity properties of a graph are both theoretically interesting and practically useful. For general k, the fastest algorithm to test for k-vertex connectivity runs in time $O(\min(kn^2 + k^4 n, k^2 n^2))$ [HRG 96]. The problem of counting the number of k-vertex cuts in a k-connected graph, for general k, however, is #P-complete [CT 98]. For $k \leq 3$, linear time algorithms are known for testing k-vertex-connectivity and for finding the vertex cuts of size $k-1$. (The cases $k = 1, 2$ are easily solved using depth-first search. For $k = 3$, see [HT 73] and [GM 00].) It is not known whether there exists a linear time algorithm to test for 4-connectivity, or to find the set of 3-vertex cuts. Such an algorithm would be useful, for instance, in efficiently solving the 2-linkage problem, in which a graph and four terminal vertices s_1, s_2, t_1 and t_2 in it are specified and we are asked to determine whether there are disjoint paths linking s_1 to t_1 and s_2 to t_2. By a theorem in [Sey 80] (see also [Sh 80] and [Th 80]), a graph has such a linkage unless it can be drawn in the plane with the vertices s_1, s_2, t_1, t_2 appearing on the infinite face in that order, or if there exists a 3-vertex cut separating the terminals from some vertex of the graph. Testing for a planar embedding as above (which is clearly an obstruction to linkage) can be done in linear time, but as for checking the connectivity requirement, it is not clear how to detect even one such 3-vertex cut in linear time.

*rdh@math.gatech.edu; School of Mathematics, Georgia Institute of Technology, Atlanta, GA 30332, USA. Partially supported by ONR project number N00014-00-1-0608 and by NSF under Grant No. DMS-9970514.

In this paper, we consider those vertex cuts that not only disconnect the graph, but do so into at least three components. More precisely, a *shredder* in a graph G is a vertex cut S such that $G \setminus S$ has at least three components. A shredder having k vertices is called a *k-shredder*. For instance, if G is a tree, the 1-shredders of G correspond precisely to the vertices of degree at least three. One application of shredders is in node connectivity augmentation; (see [CT 98]), for instance. [CT 98] also presents an algorithm to find, for general k, the set of k-shredders of a k-vertex connected graph on n vertices that runs in time $O(k^2n^2 + k^3n^{1.5})$.

This paper presents an algorithm to find the set of 3-*shredders* of a 3-*vertex connected graph* in time proportional to the number of vertices and edges in the graph. (The connectivity assumption is without loss of generality, because of the tri-connectivity algorithm in [HT 73].) The best known current bound for this problem is $O(n^2)$, which follows from the general k-shredders algorithm in [CT 98].

The motivation to study this problem came from the even directed cycle problem. Given a digraph D, the question is to decide whether D has a directed cycle of even length. This is equivalent (see [VY 89]) to several other problems of interest, for instance: Given a 0-1 square matrix A, can some of the 1's be changed to -1's in such a way that the permanent of A equals the determinant of the modified matrix (Pólya's permanent problem, [Pól 13])? When does a bipartite graph have a "Pfaffian orientation" (see [Kas 63] and [Kas 67])? When is a square matrix *sign-nonsingular* (see [BS 95] and [KLM 84])?

For the version above that is phrased in terms of Pfaffian orientations, [RST 99] presents an $O(n^3)$ algorithm, based on a structural characterization of bipartite graphs that possess a Pfaffian orientation. The exact definition of a Pfaffian orientation is not important here; what is relevant is that the structural characterization is in terms of a "trisum" operation that pastes *three* smaller graphs along a cycle on 4 vertices. With careful implementation, the running time of that algorithm can be reduced to $O(n^2)$, but attempts at further improvements run into serious difficulty. In order to take advantage of the structure theorem of [RST 99], one needs to be able, at the very least, to efficiently decide whether a 4-connected bipartite graph has a 4-shredder. It is not clear to us whether the bipartite-ness would help. Given that the corresponding problem for 3-shredders in 3-connected graphs was not known, we started with that as the first step. It should be noted that the 4-connected graphs in the above application have $O(n)$ edges, so a linear ($O(n+m)$) running time would indeed be an improvement over $O(n^2)$.

2 Notation

Given a simple undirected graph G with n vertices and m edges, we can test for 3-connectivity in linear ($O(n + m)$) time using the algorithm in [HT 73] (see also [GM 00].) The 3-shredders algorithm will then proceed in several steps. In the first step, we generate a certain set of triples of vertices that includes all the 3-shredders. In subsequent steps, we eliminate those triples that are not 3-shredders. The basic strategy for these steps is

depth-first search (dfs). We first find a depth-first spanning tree T, starting at an arbitrary vertex, which will henceforth be called the root. The edges of T (tree edges) will be directed from parent to child, and the remaining edges (back edges) will be directed from descendant to ancestor. We denote tree edges by $u \to v$ and back edges by $u \hookrightarrow v$. The adjacency list Adj(u) denotes the set of all edges with tail u. Adj$^R(u)$ will denote the set of *back edges* with head u. (We consider the tree T to be "growing downwards", with the root being the top vertex, and the children of a vertex u being listed from left to right according to their order in Adj(u).) The generation procedure and the subsequent steps will have the general format given in Table 1.

```
A: (statement to be inserted here)
dfs_step(root);

procedure dfs_step(u)
begin
    for e ∈ Adj(u) do begin
        forward_visit(e)
        if e is a tree edge u → v then begin
            dfs_step(v)
            backward_visit(e)
        end
    end
    B: (statement to be inserted here)
end
```

Table 1: General Format for the Dfs-based Steps

It is important that during the generation procedure and the subsequent steps, the edges in an adjacency list are processed in a specific order. Before we describe what the order is, we need to define the quantities $HIGH1$, $HIGH2$ and $HIGH3$.

For $v \in V$, let $D(v)$ be the set of descendants of v in the depth-first spanning tree (including v itself), and let $ND(v) := |D(v)|$. If e is a back edge $u \hookrightarrow v$, we define $HIGH1(e) = v$ and $HIGH2(e) = HIGH3(e) = u$. Now let e be a tree edge $u \to v$. We call a vertex a an *attachment* of the subtree $D(v)$ if it is a proper ancestor of u and $v' \hookrightarrow a$ for some $v' \in D(v)$. $HIGH1(e)$ is defined as the attachment of $D(v)$ that is highest in the tree, i.e. closest to the root. If no such attachment exists, we define $HIGH1(e) = \infty$. $HIGH2(e)$ is defined as the second highest attachment of $D(v)$ (∞ if no such attachment exists). Finally, $HIGH3(e)$ is defined as the third highest attachment of $D(v)$ (∞ if no such attachment exists.) Note that since G is 3-connected, $HIGH1(u \to v) \neq \infty$ unless u is the root and v is its (unique) child. Similarly, $HIGH2(u \to v) \neq \infty$ unless u is the root or its child, and v is its (unique) child.

For a vertex v that is not the root, we denote by $HIGH1$(v) the value of $HIGH1$ for

the (unique) tree edge $u \to v$. The quantities $HIGH1$, $HIGH2$ and $HIGH3$ can be easily computed in a bottom-up fashion by a dfs.

We are now ready to describe the order on Adj(u). An edge e will precede an edge f in Adj(u) if either $HIGH1(e)$ is higher in the tree than $HIGH1(f)$, or $HIGH1(e) = HIGH1(f)$ and $HIGH2(e)$ is higher than $HIGH2(f)$, or $HIGH1(e) = HIGH1(f)$ and $HIGH2(e) = HIGH2(f)$ and $HIGH3(e)$ is higher than $HIGH3(f)$. (∞ is considered higher in the tree than any vertex. Ties are broken arbitrarily in the above order.) The adjacency lists can be sorted in $O(n+m)$ time using a slight modification of radix sort with $n+1$ buckets, and future depth-first searches will use this ordering to process the edges in an adjacency list.

The vertices are then numbered 1 through n in the order in which they are *last* examined by a dfs (using the new ordering on the adjacency lists.) Henceforth, we will identify the vertices with their "last visit number" as given above. Note that the numbering *respects height on the tree* i.e. if u is an ancestor of v ($u \to^* v$), then $u \geq v$. (We use "$\to^*$" to denote a path of 0 or more tree edges.)

The first edge in an adjacency list is called a *leftmost edge*. We call v a *leftmost vertex* if it is not the root and the (unique) tree edge $u \to v$ is leftmost. Otherwise, we call v non-leftmost. A path consisting of leftmost edges is called a leftmost path. If $u \to^* v$ is a leftmost path, v is called a leftmost descendant of u.

We need to define two more quantities, $LOW1$ and RCH (for "reach"), as follows. Let $e = (u \to v)$ be a tree edge. We define $LOW1(e)$ to be the *lowest* attachment of $D(v)$ distinct from u. (By lowest, we mean farthest from the root.) If no such attachment exists, define $LOW1$ to be 0. Note that if the graph is 2-connected, every edge $u \to v$ has a non-zero $LOW1$ value unless u is the root and v its (unique) child. Section 4.1 describes how to compute $LOW1$ for all tree edges using a Union-Find procedure.
Further, we define, for every vertex u, $RCH(u) = \min\{w \mid w \hookrightarrow u\}$, where the minimum is ∞ if the set is empty. It is easy to compute $RCH(u)$ for all vertices u in a bottom-up fashion in a dfs.

Next we need a few observations about the possible arrangement of the vertices of a 3-shredder in G, with respect to the dfs tree. It can be seen that if three vertices a, b and c are not mutually comparable under the ancestor relation (corresponding to the dfs tree), i.e. they are not all on a (directed) path of tree edges, then $V(G) \setminus \{a, b, c\}$ has at most two components. (This follows from the 3-connectivity of G.) It follows that a 3-shredder in G is always of the form (a, b, c) with $a \to^* b \to^* c$. Henceforth, we shall consider a potential 3-shredder as an ordered triple of vertices, where the ordering refers to that along the tree path. Now let (a, b, c) be *any* triple of distinct vertices with $a \to p \to^* b \to q \to^* c$. The only potential components of $V(G) \setminus \{a, b, c\}$ can be listed as follows (refer to Figure 1.)

- $\mathcal{A} = A \cup A' \cup A''$, where
  A consists of $V(G) \setminus D(a)$ along with all subtrees of the form $D(v)$ where v is a child of a different from p
  A' consists of all subtrees of the form $D(v)$ where v is a child of b different from q

with $HIGH1(v) > a$
A'' consists of all subtrees of the form $D(v)$ where v is a child of c with $HIGH1(v) > a$

- $\mathcal{B} = B \cup B' \cup B''$, where
  B consists of $D(p) \setminus D(b)$
  B' consists of all subtrees of the form $D(v)$ where v is a child of b different from q with $HIGH1(v) \leq a$
  B'' consists of all subtrees of the form $D(v)$ where v is a child of c with $(b < HIGH1(v) < a)$ OR $(HIGH1(v) = a$ AND $b < HIGH2(v) < a)$

- $\mathcal{C} = C \cup C'$, where
  C consists of $D(q) \setminus D(c)$
  C' consists of all subtrees of the form $D(v)$ where v is a child of c with $(HIGH1(v) \leq b)$ OR $(HIGH1(v) = a$ AND $c < HIGH2(v) < b)$ OR $(HIGH1(v) = a$ AND $HIGH2(v) = b$ AND $c < HIGH3(v) < b)$

- $\mathcal{D}$, which consists of subtrees of the form $D(v)$ where v is a child of c with $HIGH1(v) = a$, $HIGH2(v) = b$ and $HIGH3(v) = \infty$. These subtrees form independent components of $V(G) \setminus \{a, b, c\}$ by themselves, and we'll refer to these as the *singular* components of the triple (a, b, c)

Note that $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$ and the singular components are all connected subsets of $V(G) \setminus \{a, b, c\}$, and that they partition $V(G) \setminus \{a, b, c\}$.

A triple as above (respectively, a shredder) that has a singular component is called a *singular* triple (or shredder). Conversely, a triple (shredder) that does not have any singular components is called *non-singular*. Further, a singular triple (shredder) is called *degenerate* if $HIGH1(q) \leq a$, and *non-degenerate* otherwise. Note that for a non-singular triple (a, b, c), $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$ are the only possible components of $G \setminus \{a, b, c\}$, so if (a, b, c) is to be a 3-shredder, then there must not be any edge between the vertex sets $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$. By "X-Y edge", for $X, Y \subseteq V(G)$, we mean an edge with one end in X and the other end in Y, disregarding the direction that we are associating with the edges.

The above decomposition of $V(G) \setminus \{a, b, c\}$ divides the set of proper descendants of c into A'', B'', C' and $\mathcal{D}$. The ordering of Adj(c) (in particular, the ordering of c's children in the list) implies that the subtrees in A'' occur before all the other subtrees. We define the "corner" vertex α for the triple (a, b, c) as the "lower left corner" of the first subtree not in A'', i.e. α is the lowest numbered vertex, among the descendants of c, that is not in A''. More precisely, if there exists a child v of c with $HIGH1(v) \leq a$, let v_0 be the first[1] such v and $\alpha = v_0 - ND(v_0) + 1$, otherwise $\alpha = c$. It follows that $A'' = [c - ND(c) + 1, \alpha)$.

We define corner(e) for an edge $e \in$ Adj(u) as follows: let $e' = u \to v'$ be the first *tree* edge to follow e in Adj(u) (if e is itself a tree edge, $e' = e$). We set corner(e) $= v' - ND(v') + 1$ (If no such v' exists, we set corner(e) to u.) The idea here is that, in the generation step, when we explore an edge $e \in$ Adj(u), we generate part of a triple with $c = u$ and

[1] "first" refers to the usual ordering of the adjacency lists

$a = HIGH1(e)$. The value of corner(e), as defined above, gives us the right value of α for

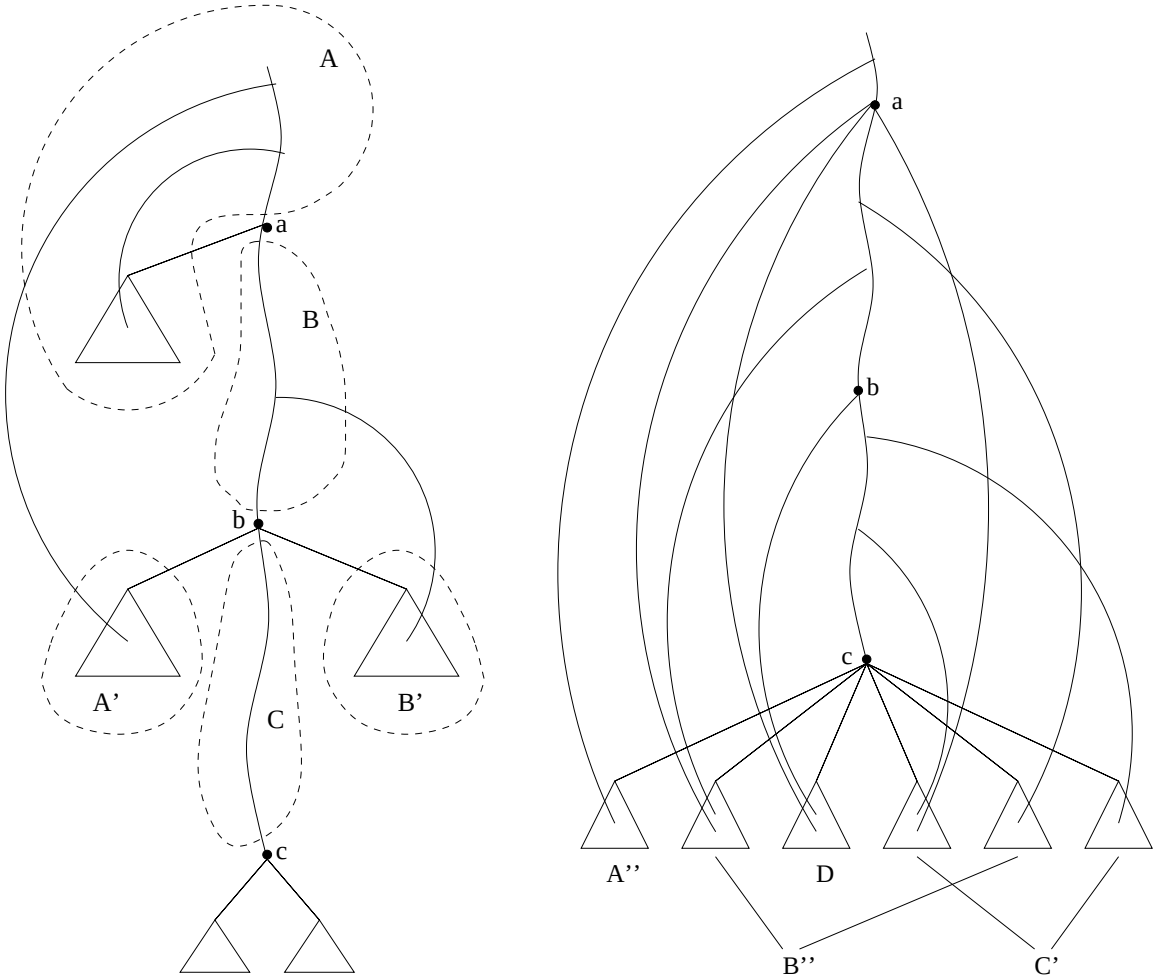

Figure 1: Potential Components in $V(G) \setminus \{a, b, c\}$

this triple. (Note that the definition of α for a triple (a, b, c) does not involve b.)

We will use the following basic lemmas about any triple (a, b, c) of vertices with $a \to p \to^* b \to q \to^* c$.

Lemma 2.1 *If the path $p \to^* b$ is not leftmost, then there is an A-B edge. In particular, for a non-singular shredder (a, b, c), the path $p \to^* b$ is leftmost.*

Proof. Suppose the path $p \to^* b$ is not leftmost, and let $s \to t$ be the first non-leftmost edge in it. Now $HIGH1(p) > a$ since a is not a cutvertex, and since s is a leftmost descendant of p, $HIGH1(s) > a$. (In general, if y is a leftmost descendant of x, it is easy to see that

$HIGH1(y) = HIGH1(x)$, assuming $HIGH1(x)$ is defined i.e x is not the root.) Let e be the first edge in Adj(s). It follows that $HIGH1(e) > a$, which means there is an A-B edge. The latter inference about non-singular shredders follows from the fact that for a non-singular shredder (a, b, c), $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$ are the only possible components of $G\backslash\{a, b, c\}$. □

Lemma 2.2 *Either $HIGH1(c) > a$, or there is an $\mathcal{A}$-$\mathcal{B}$ or $\mathcal{A}$-$\mathcal{C}$ edge. In particular, for a non-singular shredder (a, b, c), $HIGH1(c) > a$.*

Proof. If $HIGH1(c) \leq a$, then c cannot be adjacent to any vertex in $\mathcal{A}$, and hence $\mathcal{A}$ is not a component of $G\backslash\{a, b, c\}$ by itself. It follows that there must be an $\mathcal{A}$-$\mathcal{B}$ or $\mathcal{A}$-$\mathcal{C}$ edge. □

Lemma 2.3 *If the path $q \rightarrow^* c$ is not leftmost, then either $HIGH1(q) \leq a$ or there is an A-C edge. In particular, for a non-singular shredder (a, b, c), the path $q \rightarrow^* c$ is leftmost.*

Proof. The proof of the first statement is similar to the proof of Lemma 2.1. For the latter inference, note that for a non-singular shredder (a, b, c), it follows, from Lemma 2.2, that $HIGH1(c)$ (and hence $HIGH1(q)$) must be greater than a, and hence the path $q \rightarrow^* c$ is leftmost. □

For the following lemmas, let (a, b, c) be a non-singular shredder.

Lemma 2.4 *B'' is non-empty or there is a back edge $c \hookrightarrow v$ with $b < v < a$.*

Proof. This follows from the fact that c must be adjacent to a vertex in $\mathcal{B}$. □

Lemma 2.5 *One (or both) of the following conditions must hold:*

(i) $a = HIGH1(e)$ for some edge e in Adj(c)

(ii) $\exists u$ on the path $q \rightarrow^* c$, $u \neq c$, s.t. $a = HIGH1(e)$ for some non-leftmost edge e in Adj(u).

Proof. This follows from the fact that a must be adjacent to a vertex in $\mathcal{C}$. □

Lemma 2.6 *One (or both) of the following conditions must hold:*

(i) $\exists\ e \neq (b \rightarrow q)$ in Adj(b) s.t. $HIGH1(e) > a$. (In particular, if $b \rightarrow q$ is not leftmost, then this condition is automatically satisfied.)

(ii) $\exists v \in A''$ with $v \hookrightarrow b$ (i.e. b "sees" a back edge from a vertex in A''). In particular, if $b \rightarrow q$ is a leftmost edge, i.e. if $b \rightarrow^* c$ is a leftmost path, then this condition is equivalent to saying $RCH(b) < \alpha$.

Proof. This follows from the fact that b must be adjacent to a vertex in $\mathcal{A}$. □

Finally, we need the following lemma about degenerate shredders.

Lemma 2.7 *If (a, b, c) is a (singular and) degenerate shredder, then it has no $\mathcal{A}$-$\mathcal{C}$ edges, and it must have an $\mathcal{A}$-$\mathcal{B}$ edge.*

Proof. By definition, a degenerate shredder has $HIGH1(c) \leq HIGH1(q) \leq a$. It is easy to see then that there cannot be any $\mathcal{A}$-$\mathcal{C}$ edges, and that c cannot be adjacent to any vertex in $\mathcal{A}$. It follows that $\mathcal{A}$ cannot be a component of $G \setminus \{a, b, c\}$ by itself, and hence there must be an $\mathcal{A}$-$\mathcal{B}$ edge. □

3 The Generation Step

As mentioned before, the generation step follows the general format given in Table 1. The pseudo-code for replacing the lines forward_visit(e) and backward_visit(e) is given in tables 2 and 3 respectively.

A singular shredder (a, b, c), by definition, has a singular component. Hence there is an edge $e = (c \to v)$ in Adj(c) such that $HIGH1(e) = a$, $HIGH2(e) = b$ and $HIGH3(e) = \infty$. Hence we can generate the triple when we are about to explore e. In order to find out whether the triple is degenerate or not, we need to know what q is, i.e. we need to know which child of b is currently active (in the dfs). We keep track of this information in the array active_child, which is updated whenever a recursive call is made. The rest of the section gives an informal description of how the generation step finds non-singular shredders, before giving proofs of correctness and the time bound for the generation step.

Let (a, b, c) be a non-singular shredder. By Lemma 2.5, $a = HIGH1(e)$, where e is as in that lemma. We will generate "candidate pairs" (a, c) and find the corresponding vertex b later. The candidate pairs will be stored in a data structure that we call PSTACK. This will be a stack of "blocks", separated by end-markers, similar to the stack in a recursion. The individual blocks will be ordered lists of candidate pairs. Before each call dfs_step(v) for a *non-leftmost vertex* v, an end-marker is inserted on top of PSTACK, signifying that a fresh block is now on top of PSTACK. After the exit from dfs_step(v), the topmost block (and the end-marker) are removed from the PSTACK. The first step of generating a triple (a, b, c) begins when some edge e in Adj(c) is explored, at which point $HIGH1(e)$ will be our guess for a, which might be revised later. In addition, the value of α for this "candidate pair" is set to corner(e). This candidate pair (a, c) is added to the beginning of the current (topmost) block of PSTACK, and will be removed from PSTACK either to be moved to a list of triples (when the vertex b is detected), or to be discarded. As the dfs backs up over the tree path $a \to^* c$, we expect to recognize a vertex on the path as the right "b" for the pair. There are three situations in which we realize that we have come across b, corresponding to the conditions in Lemma 2.6:

1. Whenever we back up over a non-leftmost edge $u \to v$, we mark u as the vertex b for all the candidate pairs in the current block of PSTACK (and move them to a list of triples). In particular, the topmost block of PSTACK (and the end-marker below it) are removed.

2. Whenever we explore a non-leftmost edge e in Adj(u) and see a candidate pair (a, c) in the current block of PSTACK with $a < HIGH1(e)$, we create the triple (a, u, c).

3. Whenever we back up over a leftmost edge $u \rightarrow v$, and see a candidate pair (a, c) in the current block of PSTACK with $RCH(u) < \alpha$, we create the triple (a, u, c) and remove the pair (a, c) from PSTACK.

In situation 2 above, in addition to generating the corresponding triple, we also *revise* the value of a in the pair to $HIGH1(e)$ (the revised pair stays on PSTACK). This corresponds to Lemma 2.5(ii). A pair that has already been revised once, however, is discarded. This is essential for keeping the overall time taken for the PSTACK operations linear.

```
1  if e = (u → v) AND HIGH3(v) = ∞ then    comment generating a singular triple
2       generate the triple (a = HIGH1(v), b = HIGH2(v), c = u) and mark it non-
        degenerate or degenerate depending on whether HIGH1(q) > a or not (where
        q = active_child(b))
3  if e is non-leftmost then begin
4       let (a, c) be the first candidate pair in the current block of PSTACK (set it to null
        if the end-marker is encountered instead)
5       while (a, c) is not null AND a < HIGH1(e) do begin
6            create the triple (a, b=u, c)
7            if the pair (a, c) is unrevised then
8                 set a = HIGH1(e) and mark the pair (a, c) as revised
9            else discard the pair (a, c) from PSTACK
10           set (a, c) to the next pair in the current block (set it to null if the end-marker
             is encountered instead)
11      end
12      add the pair (a=HIGH1(e), c=u) to the beginning of the current block; set α =
        corner(e)
13      if e is a tree edge then add an end-marker on top of PSTACK
14 end
15 if e is a tree edge then set active_child(u) = v
```

Table 2: Generation step: pseudo-code for forward_visit(e)

If u is the current vertex in the dfs, and (a, c) is a candidate pair in the current block, then c would be a leftmost descendant of u, and a would be an ancestor of u. Thus the pairs in the current block consist of vertices that are all on a tree path containing u. Moreover, the pairs $(a_1, c_1), (a_2, c_2), \ldots$ in the block (in order) will be such that the vertices $\ldots, c_2, c_1, u, a_1, a_2, \ldots$ appear on this path in the order listed, and that $\ldots \leq \alpha_2 \leq \alpha_1$. This is essential for efficiently updating the pairs on PSTACK.

Theorem 3.1 *The generation step runs in time $O(n + m)$ and therefore the number of triples generated is also $O(n + m)$.*

```
1  if e = (u → v) is non-leftmost then    comment backing up over a non-leftmost edge
2       move all pairs in the current block to a list of triples, setting b = u; remove the
        end-marker
3  else   comment backing up over a leftmost edge
4       while the pair (a, c) in the beginning of the current block has α > RCH(u) do
5            remove the pair from PSTACK and create the triple (a, b = u, c)
```

Table 3: Generation step: pseudo-code for backward_visit(e)

Proof. Since the generation step has the format given in Table 1, we only need to verify that the **while** loops in the pseudo-code for forward_visit and backward_visit (line 5 of Table 2 and line 4 of Table 3 respectively) and line 2 of Table 3 take $O(n+m)$ time overall. The total number of distinct candidate pairs processed on PSTACK is at most $2m$, since each edge leads to the generation of at most one candidate pair, and this pair may be revised only once. Since the time taken by the **while** loops and line 2 of Table 3 is at most the number of distinct candidate pairs plus the number of edges, it follows that the overall time taken is $O(n+m)$. □

Before the next lemma, we need a definition. For a vertex u, the (maximal) tree path joining the root to the unique leaf that is a leftmost descendant of u is called the *canonical path* containing u.

Lemma 3.2 *During the generation step, the following condition holds immediately before the* ***while*** *loops in forward_visit(e) and backward_visit(e) (line 5 of Table 2 and line 4 of Table 3 respectively). If u is the current vertex in the search, then the pairs in the current block of PSTACK consist of vertices that are all on the canonical path containing u. Moreover, the pairs $(a_1, c_1), (a_2, c_2), \ldots$ in the block (in order) will be such that the vertices $\ldots, a_2, a_1, u, c_1, c_2, \ldots$ appear on this path in the order listed (possibly with repetion), and $\ldots \leq \alpha_2 \leq \alpha_1$.*

Proof. We shall proceed by induction, proving that all the operations that change the current vertex, or the current block of PSTACK (or both) preserve the properties stated in the lemma. Suppose that at some time instant in the generation step, the pairs in the current block satisfy the assertion of the lemma. Let these pairs (in order) be $(a_2, c_2), (a_3, c_3), \ldots$, and let u be the current vertex. Suppose the search is exploring a non-leftmost edge e in Adj(u) with $HIGH1(e) = a_1$. Since the revision process (the **while** loop on line 5, Table 2) either discards pairs with $a < a_1$ or revises them (by setting $a = a_1$), it is clear that it preserves the required property of PSTACK. If the generation step then adds the pair $(a_1, c_1 = u)$ while exploring an edge in Adj(u), it is easy to see that $\alpha_2 \leq \alpha_1$, and that $a_1 \leq a_2$ because of the revision process. Thus the assertion of the lemma still holds after this pair is added. Now suppose the recursive call dfs_step(v) (that makes v the current vertex) is made after the tree edge $u \to v$ is explored (i.e. after forward_visit($u \to v$).) If $u \to v$ is non-leftmost, then a fresh block would have been started before the recursive call, so the

lemma holds trivially immediately after the recursive call. If $u \to v$ is leftmost, then no candidate pair with $c = u$ is yet on PSTACK, since pairs are generated only while exploring non-leftmost edges. Hence the lemma still holds immediately after the recursive call. While backing up over a non-leftmost edge $u \to v$, the current block is removed, restoring PSTACK to its state just before the recursive call dfs_step(v), and hence the lemma still holds by induction. Finally, suppose that the search is backing up over a leftmost edge $u \to v$, with (a, c) being the first pair in the current block. It suffices to verify that a is a proper ancestor of u (unless $a = u$ is the root) to prove that the lemma still holds after the search backs up over $u \to v$. Since the pair (a, c) has survived on PSTACK till the search has backed up over to the vertex u, we claim by induction that there is no edge between $A \cup A''$ and $D(v) \setminus (A'' \cup \{c\})$, where A is defined for the pair (a, c) analogous to the definition for triples, and $A'' = [c - ND(c) + 1, \alpha)$. If there is such an edge, say between A'' and $D(v) \setminus (A'' \cup \{c\})$, it would be detected in the **while** loop in Table 3. If there is an edge between A and $D(v) \setminus (A'' \cup \{c\})$, then either there is a non-leftmost edge in the path $v \to^* c$ (which will be detected by line 2 in Table 3), or $\exists v'$ on the path $v \to^* c$, $v' \neq c$, such that $HIGH1(e) > a$ for some non-leftmost edge e in Adj(v') (which will be detected in the **while** loop in Table 2.) This proves the claim and implies that a must be a proper ancestor of u (unless $a = u$ is the root), otherwise $A \cup A''$ and $D(v) \setminus (A'' \cup \{c\})$ would be (non-empty) components of $G \setminus \{a, c\}$. This proves the lemma. □

Theorem 3.3 *The generation step finds a (multi-)set of triples that includes the set of 3-shredders of G. Furthermore, the non-singular triples generated have no $\mathcal{A}$-$\mathcal{C}$ edges, the path $q \to^* c$ is leftmost for all such triples, and they satisfy the properties given in Lemma 2.5.*

Proof. First, let (a, b, c) be a singular shredder and $D(v)$ be a singular component of (a, b, c). Then the triple (a, b, c) will be generated when the edge $c \to v$ is explored.

Consider a non-singular shredder (a, b, c). Let e_1 be the first edge in Adj(c) with $b < HIGH1(e_1) \leq a$, and let $HIGH1(e_1) = a_1$. (We know such an edge exists because of Lemma 2.4.) Now $HIGH1(e_1) \leq a < HIGH1(c)$ by Lemma 2.2; in particular, e_1 is non-leftmost. It follows that the candidate pair (a_1, c) is generated when e_1 is explored (line 12 of Table 2), with the correct value of α. Furthermore, if $a_1 \neq a$, then by Lemma 2.5, there is a vertex u on the path $q \to^* c$, $u \neq c$, such that $a = HIGH1(e_2)$ for some non-leftmost edge e_2 in Adj(u). Subject to the above condition, choose u to be closest to c and subject to this, choose e_2 to be the earliest in Adj(u). Since the shredder (a, b, c) does not have any A-C or B-C edges, that candidate pair (a_1, c) remains unrevised till the edge e_2 is explored, at which point the pair will be revised with a_1 being changed to a. (The fact that the revision process reaches the pair follows from Lemma 3.2.) Again, since (a, b, c) has no $\mathcal{A}$-$\mathcal{C}$ edges, it follows that the pair (a, c) then stays on PSTACK until the search backs up to the vertex b. By Lemma 2.6, the vertex b is detected by one of the two **while** loops in tables 2 and 3, or line 2 in Table 3. (Again, the fact that b is detected as above follows from Lemma 3.2.) Thus the triple (a, b, c) will be generated.

Also, if a triple (a, b, c) is generated from a candidate pair (a, c), the fact that the candidate pair survived on PSTACK till the search backed up over the edge $b \to q$ means

that there is no edge between $A \cup A''$ and $D(q) \setminus (A'' \cup \{c\})$, where $A'' = [c - ND(c) + 1, \alpha)$. (This is simply the claim proved at the end of the proof of Lemma 3.2.) In particular, the non-singular triples generated have no $\mathcal{A}$-$\mathcal{C}$ edges. Similarly, it follows that the path $q \rightarrow^* c$ is leftmost for all the non-singular triples generated. To see that Lemma 2.5 holds for the non-singular triples, note that if a triple is generated from an unrevised candidate pair, then Lemma 2.5(i) is satisfied, and if a triple (a, b, c) is generated from a candidate pair (a, c) that had been revised at a vertex u, then b must be a proper ancestor of u, i.e. Lemma 2.5(ii) is satisfied. □

The next section describes some intermediate computation that needs to be done before we move on to the detection steps.

4 Intermediate computation

Before the detection steps described in Section 5, we eliminate multiple copies of non-singular and singular triples generated by the generation step. From the proof of Theorem 3.3, it can be seen that for non-singular triples, among multiple copies of a triple, the one with the lowest value of α should be retained. This can be efficiently done by simultaneously sorting (using radix sort) for all vertices u, the lists of triples with $c = u$ in lexicographic order of (a, b, α), and then scanning the lists for multiple entries. Similarly, if a triple (a, b, c) is generated both as a non-singular and a singular triple, then the non-singular triple is discarded. Furthermore, non-singular triples that do not satisfy the property $HIGH1(c) > a$ (as in Lemma 2.2) are discarded, and triples with $\mathcal{A}$,$\mathcal{B}$ or $\mathcal{C}$ empty (i.e. a = root or a is the parent of b or b is the parent of c, respectively) are discarded.

4.1 Computing $LOW1$

This section describes how to compute $LOW1$ for all the tree edges in linear time, using a Union-Find procedure. Let $z \hookrightarrow u$ be a back edge such that no proper descendant of u is the head of any back edge, and let $u \rightarrow v \rightarrow w \rightarrow^* z$ be the tree path from u to z. It is easy to see that for for all tree edges e along the path $v \rightarrow^* z$, $LOW1(e) = u$. If we now discard the back edge $z \hookrightarrow u$ and consider another back edge as above, we can compute, in the same way, $LOW1$ for other tree edges for which the quantity has not yet been computed. Proceeding in this fashion, we can compute $LOW1$ eventually for all tree edges. However, in order to not examine edges repeatedly (and hence spend too much time), we need to contract a tree edge once its $LOW1$ value is computed. It can be seen that in the situation above, *contracting* any edge on the path $v \rightarrow^* z$ keeps the $LOW1$ values of the remaining tree edges the same. (By contracting a tree edge $x \rightarrow y$, we mean removing the edge and the vertex y, and replacing the end y in any other (tree and back) edges with the vertex x.) In order to pick the back edges in the manner described above, all the back edges are collected in a list, sorting the edges $z \hookrightarrow u$ in increasing order of u. The back edges are then picked from this list, in order, computing $LOW1$ for the relevant tree edges as above

and contracting them. Note that contracting those edges does not change the head of any other back edge, and hence does not affect the ordering of the list of back edges.

The edge contraction is implemented symbolically using Union-Find. A Union-Find algorithm implements a data structure for a partition of a ground set (the set of vertices in this case). Each set has a *representative* that is used to refer to the set. The Union-Find algorithm supports two operations:
union(x,y), which forms the union of the sets containing x and y (destroying the original sets), and
find-set(x), which returns the representative of the set containing x.
As the contractions are carried out, the vertex set is dynamically represented as a partition of the original vertex set. Initially all vertices are in singleton sets. In general, the vertex represented by x is the representative of the set containing x.

Contraction of an edge $x \to y$ is done by merging the sets containing x and y, setting the representative of the merged set to that of the former set. (If the Union-Find algorithm does not implement the latter requirement, we can implement an additional function on the vertex set that maps the representative assigned by the algorithm to the one desired.) Computing the $LOW1$ values as above requires $O(m)$ operations (union and find-set) to be performed on a ground set of size n. Note also that the unions all correspond to edges in the dfs tree (i.e. they are all of the form union(x,y) where x and y are adjacent in the dfs tree.) Hence this is a "graphical Union-Find" where the graph is actually a tree. This special case of Union-Find can be solved in time $O(m+n)$ on a random-access machine (see [GT 85] or [J 98].) The classical algorithm for Union-Find (based on "weighted-union of trees with finds executing path compression"; see [T 75]) performs the above Union-Find in time $O((m+n)\alpha(m+n))$, where α is the functional inverse of an Ackermann-like function.

4.2 The "corner" vertices for a triple

The vertex α for a triple (a, b, c) was defined in such a way that the interval $[c-ND(c)+1, \alpha)$ is precisely A''. Similarly, we define the other corner vertices β, γ, δ and ϵ for demarcating B'', C' and $\mathcal{D}$, as follows:

β: If there exists a child v of c with $(HIGH1(v) = a$ AND $HIGH2(v) \leq b)$ OR $HIGH1(v) < a$, let v_0 be the first such v and $\beta = v_0 - ND(v_0) + 1$
otherwise $\beta = c$

γ: If there exists a child v of c such that $(HIGH1(v) = a$ AND $HIGH2(v) < b)$ OR $(HIGH1(v) = a$ AND $HIGH2(v) = b$ AND $HIGH3(v) < b)$ OR $HIGH1(v) < a$, let v_0 be the first such v and $\gamma = v_0 - ND(v_0) + 1$
otherwise $\gamma = c$

δ: If there exists a child v of c with $HIGH1(v) < a$, let v_0 be the first such v and $\delta = v_0 - ND(v_0) + 1$
otherwise $\delta = c$

ϵ: If there exists a child v of c with $HIGH1(v) \leq b$, let v_0 be the first such v and $\epsilon = v_0 - ND(v_0) + 1$
otherwise $\epsilon = c$

Note that $A'' = [c - ND(c) + 1, \alpha)$, $B'' = ([\alpha, \beta) \cup [\delta, \epsilon))$, $C'' = ([\gamma, \delta) \cup [\epsilon, c))$,and $\mathcal{D} = [\beta, \gamma)$.

4.3 Computing the corner vertices for the triples

For the following discussion, list(u), for all vertices u, is the list of triples (a, b, c) with $c = u$.

computing ϵ
sort list(u) in decreasing order of b $\forall u \in V(G)$
for $u \in V(G)$ **do begin**
for tree edge $e \in$ Adj(u) **do**
mark off triples from list(u) with $\epsilon =$ corner(e) (and remove them from the list) until a triple with $b < HIGH1(e)$ is encountered
set $\epsilon = u$ for all triples left over in list(u)
end

computing δ
sort list(u) in decreasing order of a $\forall u \in V(G)$
for $u \in V(G)$ **do begin**
for tree edge $e \in$ Adj(u) **do**
mark off triples from list(u) with $\delta =$ corner(e) (and remove them from the list) until a triple with $a \leq HIGH1(e)$ is encountered
set $\delta = u$ for all triples left over in list(u)
end

computing γ
sort list(u) in decreasing lexicographic order of (a, b) $\forall u \in V(G)$
for $u \in V(G)$ **do begin**
for tree edge $e \in$ Adj(u) **do**
if $HIGH3(v) = \infty$ **then**
mark off triples from list(u) with $\gamma =$ corner(e) (and remove them from the list) until a triple with ($a = HIGH1(e)$ AND $b \leq HIGH2(e)$) OR $a < HIGH1(e)$ is encountered
else
mark off triples from list(u) with $\gamma =$ corner(e) (and remove them from the list) until a triple with ($a = HIGH1(e)$ AND $b < HIGH2(e)$) OR $a < HIGH1(e)$ is encountered
set $\gamma = u$ for all triples left over in list(u)
end

computing β
sort list(u) in decreasing lexicographic order of (a, b) $\forall u \in V(G)$

for $u \in V(G)$ **do begin**
    for tree edge $e \in$ Adj(u) **do**
        mark off triples from list(u) with β = corner(e) (and remove them from the list) until a triple with ($a = HIGH1(e)$ AND $b < HIGH2(e)$) OR $a < HIGH1(e)$ is encountered
    set $\beta = u$ for all triples left over in list(u)
end

computing α (for singular triples)
    similar to computing δ; simply replace the condition $a \leq HIGH1(e)$ with $a < HIGH1(e)$.

We need to define (and compute for all the triples) an additional corner vertex to demarcate A' and B', but it can be done in a similar fashion as discussed above, and is hence omitted. Finally, in preparation for the detection steps carried out in Sections 5.2 and 5.3, we need to mark those triples in which c is not adjacent to any vertex in $\mathcal{B}$. The above happens for a triple if and only if B'' is empty (i.e. $\alpha = \beta$ and $\delta = \epsilon$) *and* there is no back-edge $c \hookrightarrow v$ with $b < v < a$. Checking the first condition for all triples is trivial. The second condition is also easily checked by looking at the first child v of c with $HIGH1(v) \leq b$ (this information can be computed for all triples along with ϵ) and then examining whether the edge e *preceding* $c \rightarrow v$ in Adj(c) has $HIGH1(e) < a$ or not.

5 Weeding out the non-shredders

The generation procedure gives us a set of triples that includes the set of 3-shredders of the graph. It remains to weed out those triples that are not 3-shredders, by recognizing those that have edges between the (potential) components $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$. A non-singular triple is discarded as soon as such a "bad" edge is found. A singular triple, on the other hand, could have these bad edges as long as it has enough singular components to make it a 3-shredder. We detect these edges in several dfs-like steps, each step dealing with a certain type or types of these bad edges. The general idea behind all the steps is the same (except for the one described in Section 5.6). The types of bad edges handled by a detection step would all have either their head or tail in one of the sets B or C; so the detection of the bad edges for a given triple is carried out while the search is inside B or C, more precisely, as the search backs up over the tree paths $p \rightarrow^* b$ or $q \rightarrow^* c$ respectively. Furthermore, the other end of each of the bad edges would be in a set of vertices that we shall call the *forbidden set*. For instance, if we are dealing with $\mathcal{A}$-$\mathcal{C}$ edges, then the bad edges would all have one end in C and the other end in $A \cup A''$. Thus the detection will be done while backing up over the path $q \rightarrow^* c$, and the forbidden set in this case will be $A \cup A''$.
The triples are stored and processed on a stack similar to the one used in the generation step. The current block of the stack would contain the triples (a, b, c) for which the current vertex u is in $p \rightarrow^* b$ or $q \rightarrow^* c$, as the case may be. We examine the back edges with head u, and for each such edges e, mark off (and remove) those triples in the current block for

which the other end of e is in the forbidden set. Also, while exploring a non-leftmost tree edge e in Adj(u), we mark off and remove those triples with $HIGH1(e)$ in the forbidden set. In order to do this step efficiently, we need to keep the forbidden sets of the triples in the current block of the stack monotone i.e. if (a_1, b_1, c_1) appears before (a_2, b_2, c_2) in the current block, then the forbidden set of (a_1, b_1, c_1) contains the forbidden set of (a_2, b_2, c_2).

The general format of the detection steps will be as given in Table 1, with statement A being replaced by the subroutine *sort_lists* and statement B being replaced by the subroutine *load_triples*(u). The subroutine sort_lists sorts *list*(u) for all vertices u, where list(u) is the list of triples with $c = u$ (or $b = u$ as the case may be). The subroutine load_triples(u) loads list(u) at the beginning of the current block of the stack while maintaining the property mentioned in the previous paragraph. The flowchart in Figure 2 describes the order in which the detection steps are applied to the triples to eventually determine the set of 3-shredders of the graph. The following subsections describe the individual steps in detail.

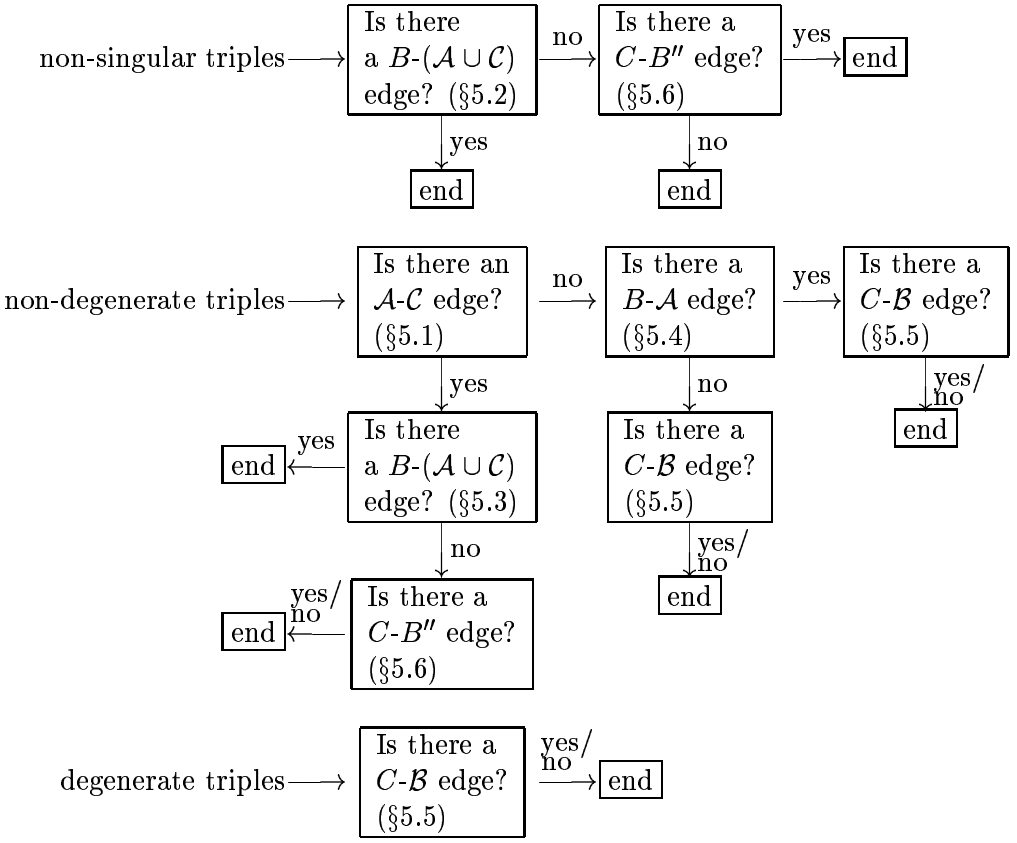

Figure 2: Flowchart describing the order of application of the detection steps

5.1 Detecting $\mathcal{A}$-$\mathcal{C}$ edges

From Theorem 3.3, the non-singular triples generated do not have any $\mathcal{A}$-$\mathcal{C}$ edges. Furthermore, degenerate triples do not have any $\mathcal{A}$-$\mathcal{C}$ edges, as $HIGH1(q) \leq a$. Hence this step is only required for singular, non-degenerate triples.

The forbidden set for a triple (a, b, c) is $A \cup A''$ and the detection is carried out on the path $q \rightarrow^* c$. For all vertices u, list(u) is the list of non-degenerate triples with $c = u$. The subroutine sort_lists sorts list(u) $\forall u \in V$ in increasing order of a. Note that this is consistent with the monotonicity required for the forbidden sets i.e. the forbidden set $(A \cup A'')$ of (a_1, b_1, u) contains the forbidden set of (a_2, b_2, u) if $a_1 \leq a_2$. A computational remark is in order here. The above sorting can be done in linear time with a radix sort with $n+1$ buckets, similar to the sorting of the adjacency lists described in Section 2. The pseudo-code for replacing forward_visit(e) and backward_visit(e) is given in tables 4 and 5 respectively.

```
1  if e is non-leftmost then begin
2       while the triple (a, b, c) in the beginning of the current block of STACK has a <
        HIGH1(e) do
3            mark the triple as having an A-C edge and remove it from STACK
4       if e is a tree edge then add an end-marker on top of STACK
5  end
```

Table 4: Detecting $\mathcal{A}$-$\mathcal{C}$ edges: pseudo-code for forward_visit(e)

```
1  remove the triples with q = v from the current block of STACK
2   if e = (u → v) is non-leftmost then begin   comment  backing up over a
                                                         non-leftmost edge
3       mark all triples (a, b, c) in the current block of STACK as having an A-C edge
4       remove the block (and the end-marker) from STACK
5  end
6  else   comment backing up over a leftmost edge
7       while the triple (a, b, c) in the beginning of the current block has α > RCH(u)
        do
8            mark the triple as having an A-C edge and remove it from STACK
```

Table 5: Detecting $\mathcal{A}$-$\mathcal{C}$ edges: pseudo-code for backward_visit(e)

Table 6 gives the pseudo-code for the subroutine load_triples(u).

Lemma 5.1 *During the detection step, the following condition holds immediately before the* ***while*** *loops (line 2 of Table 4 and line 7 of Table 5). If u is the current vertex in the search,*

```
1  while list(u) is non-empty do begin
2        let (a_1, b_1, c_1) be at the end of list(u)
3        if the current block of STACK is empty then
4              remove (a_1, b_1, c_1) from list(u) and add it at the beginning of the current block
               of STACK
5        else begin
6              let (a_2, b_2, c_2) be at the beginning of the current block of STACK
7              if c_1 > c_2 AND a_1 > a_2 then
8                    mark (a_2, b_2, c_2) as having an A-C edge and remove it from STACK
9              else remove (a_1, b_1, c_1) from list(u) and add it at the beginning of the current
               block of STACK
10       end
11 end
```

Table 6: Detecting $\mathcal{A}$-$\mathcal{C}$ edges: pseudo-code for load_triples(u)

then the triples (a, b, c) in the current block of STACK are such that a, b and c are all on the canonical path containing u, with a and b being proper ancestors of u and c a (leftmost) descendant of u. Moreover, the triples $(a_1, b_1, c_1), (a_2, b_2, c_2), \ldots$ in the block (in order) will be such that the vertices $\ldots, a_2, a_1, u, c_1, c_2, \ldots$ appear on this path in the order listed (possibly with repetition), and that $\ldots \leq \alpha_2 \leq \alpha_1$. It follows that $(A_1 \cup A_1'') \supseteq (A_2 \cup A_2'') \ldots$ and so on.

Proof. We shall use an inductive argument similar to the one used in the proof of Lemma 3.2. It is clear that forward_visit(e) and the recursive call preserve the property asserted in the lemma. After backing up a non-leftmost edge $u \to v$, the property still holds by induction because STACK is restored to its state just before the recursive call dfs_step(v). After backing up a leftmost edge, the property still holds because of line 1 in Table 5.

Suppose now that during the search, we are at the end of dfs_step(u) and want to load list(u) on STACK. We need to verify that load_triples(u) preserves the required property of STACK. The procedure load_triples(u) looks at the triple (a_1, b_1, c_1) at the end of list(u) and compares it with the triple (a_2, b_2, c_2) at the beginning of the current block of STACK. It either moves (a_1, b_1, c_1) to STACK if it finds that doing so maintains the property stated in the lemma, or discards one of the triples, marking it as having an $\mathcal{A}$-$\mathcal{C}$ edge. (This is continued until all triples in list(u) have either been loaded onto STACK or marked as having an $\mathcal{A}$-$\mathcal{C}$ edge.) If $c_1 = c_2$ (this happens if (a_2, b_2, c_2) is originally from list(u)), then it follows that (a_1, b_1, c_1) can be moved from list(u) to STACK while maintaining the property given in the lemma. If $c_1 \neq c_2$, then c_2 is a proper leftmost descendant of c_1. Also, since (a_2, b_2, c_2) is still on STACK and the search is at the end of dfs_step(c_1), it follows that b_2 is a proper ancestor of c_1. Hence $c_1 \in C_2$. Suppose $a_1 > a_2$, and let s be the leftmost child of c_1. $D(s)$ cannot be a singular component of (a_1, b_1, c_1), since it has two distinct attachments a_2 and b_2 other than a_1. But then a singular component of (a_1, b_1, c_1) gives

rise to an $\mathcal{A}$-$\mathcal{C}$ edge for (a_2, b_2, c_2), which is hence removed from STACK and marked as having an $\mathcal{A}$-$\mathcal{C}$ edge. If $a_1 \leq a_2$, it follows that (a_1, b_1, c_1) can be moved from list(u) to STACK while maintaining the property given in the lemma. □

Theorem 5.2 *The detection step correctly marks, in $O(n+m)$ time, those of its input triples that have an $\mathcal{A}$-$\mathcal{C}$ edge.*

Proof. Suppose a triple (a, b, c) is marked by the search as having an $\mathcal{A}$-$\mathcal{C}$ edge. We need to verify that it indeed has one. If the triple is marked by the subroutine load_triples, then it can be easily seen from the proof of the previous lemma that it has an $\mathcal{A}$-$\mathcal{C}$ edge. If the triple is marked by line 3 in Table 4, the edge $e = u \rightarrow v$ is such that the subtree $D(v) \subseteq C$ has an attachment higher than a, that is, in A. Hence the triple has an $\mathcal{A}$-$\mathcal{C}$ edge. If the triple is marked by line 3 in Table 5, the edge $e = u \rightarrow v$ is non-leftmost, hence by Lemma 2.3 it follows that the triple has an $\mathcal{A}$-$\mathcal{C}$ edge (since $HIGH1(q) > a$, by definition, for non-degenerate triples.) Finally, if the triple is marked by line 8 in Table 5, the vertex $u \in C$ has a back edge coming in from a vertex in A'', hence the triple has an $\mathcal{A}$-$\mathcal{C}$ edge.
Conversely, suppose a triple (a, b, c) has an $\mathcal{A}$-$\mathcal{C}$ edge. Then it has either a C-A'' edge, or a C-A edge (or both.) From the previous lemma, it then follows that in the first case, such an edge would be detected by the **while** loop on line 7 in Table 5. In the second case, the edge would be detected by the **while** loop in Table 4 or line 2 in Table 5. Hence a triple with an $\mathcal{A}$-$\mathcal{C}$ edge will be marked accordingly.
Finally, for the time bound, note that the subroutine load_triples takes $O(1)$ time per triple, and hence loading list(u) for all vertices u takes $O(n+m)$ time overall (as there are $O(n+m)$ triples.) Also, line 1 in Table 5 can be efficiently executed by maintaining, for every vertex v, the list of triples with $q = v$ (where the triple (a, b, c) is such that $a \rightarrow p \rightarrow^* b \rightarrow q \rightarrow^* c$.) The time taken for executing line 1 would then be proportional to the number of triples, and hence $O(n+m)$. □

5.2 Detecting B-$(\mathcal{A} \cup \mathcal{C})$ edges in non-singular triples

This step is only executed for non-singular triples. A triple is discarded as soon as such an edge is found. Before the detection step itself, triples (a, b, c) such that c is not adjacent to any vertex in $\mathcal{B}$ are discarded. (Note that for such triples, $\mathcal{B}$ cannot be a component of $G \setminus \{a, b, c\}$ in itself, and hence there is a $\mathcal{B}$-$(\mathcal{A} \cup \mathcal{C})$ edge. In fact, since c is not adjacent to any vertex in $\mathcal{B}$, such an edge must be a B-$(\mathcal{A} \cup \mathcal{C})$ edge.)

The forbidden set for a triple is $\mathcal{A} \cup \mathcal{C}$ and the detection is carried out on the path $p \rightarrow^* b$. For all vertices u, list(u) is the list of non-singular triples with $b = u$. The subroutine sort_lists sorts list(u) $\forall u \in V$ such that (a_1, u, c_1) precedes (a_2, u, c_2) in the list iff $a_1 < a_2$, or $a_1 = a_2$ and $c_1 \geq c_2$.

The pseudo-code for replacing forward_visit(e) and backward_visit(e) is given in tables 7 and 8 respectively.

Table 9 gives the pseudo-code for the subroutine load_triples(u).

```
1  if e is non-leftmost then begin
2       while the triple (a, b, c) in the beginning of the current block of STACK has a <
        HIGH1(e) do
3            discard the triple
4       if e is a tree edge then add an end-marker on top of STACK
5  end
```

Table 7: Detecting B-$(\mathcal{A} \cup \mathcal{C})$ edges in non-singular triples: pseudo-code for forward_visit(e)

```
1  while the triple (a, b, c) at the beginning of the current block of STACK has a = u do
2       remove the triple from STACK
3    if e = (u → v) is non-leftmost then    comment  backing up over a
                                                     non-leftmost edge
4       discard all triples (a, b, c) in the current block of STACK; remove the block (and the
        end-marker) from STACK
5  else   comment backing up over a leftmost edge
6       for edge w ↪ u in Adj^R(u) do

7            while the triple (a, b, c) at the beginning of the current block has w ∈
             (A ∪ C) do
8                 discard the triple
```

Table 8: Detecting B-$(\mathcal{A} \cup \mathcal{C})$ edges in non-singular triples: pseudo-code for backward_visit(e)

```
1  while list(u) is non-empty do begin
2       let (a1, b1, c1) be at the end of list(u)
3       if the current block of STACK is empty then
4            remove (a1, b1, c1) from list(u) and add it at the beginning of the current block
             of STACK
5       else begin
6            let (a2, b2, c2) be at the beginning of the current block of STACK
7            if b1 = b2 then
8                 if q1 ≠ q2 then
9                      if HIGH1(q1) > a2 then
10                          discard (a2, b2, c2)
11                     else remove (a1, b1, c1) from list(u) and add it at the beginning of the
                       current block of STACK
12                else if c1 = c2 then
13                     remove (a1, b1, c1) from list(u) and add it at the beginning of the cur-
                       rent block of STACK
```

```
14              else comment c_1 > c_2
15                  discard (a_2, b_2, c_2)
16          else if a_1 > a_2 then
17              if q_1 is non-leftmost then
18                  discard (a_2, b_2, c_2)
19              else if the second edge e in Adj(b_1) has HIGH1(e) > a_1 then
20                  discard (a_2, b_2, c_2)
21              else if c_1 > b_2 then
22                  discard (a_2, b_2, c_2)
23              else if c_1 = b_2 then
24                  if HIGH1(q_2) > a_1 then
25                      discard (a_1, b_1, c_1)
26                  else discard (a_2, b_2, c_2)
27              else if c_2 ≥ c_1 then
28                  discard (a_2, b_2, c_2)
29              else if RCH(b_1) < α_2 then
30                  discard (a_2, b_2, c_2)
31              else discard (a_1, b_1, c_1)
32          else if c_1 ≥ b_2 OR c_1 = c_2 then
33              remove (a_1, b_1, c_1) from list(u) and add it at the beginning of the current
                block of STACK
34          else discard (a_2, b_2, c_2)
35      end
36  end
```

Table 9: Detecting B-$(\mathcal{A} \cup \mathcal{C})$ edges in non-singular triples: pseudo-code for load_triples(u)

Lemma 5.3 *During the detection step, the following condition holds immediately before the* ***while*** *loops on line 2 of Table 7 and line 7 of Table 8. If u is the current vertex in the search, then the triples (a, b, c) in the current block of STACK are such that a and b are on the canonical path containing u, with a being a proper ancestor of u and b a (leftmost) descendant of u. Moreover, the triples $(a_1, b_1, c_1), (a_2, b_2, c_2), \ldots$ in the block (in order) will be such that the vertices $\ldots, a_2, a_1, u, b_1, b_2, \ldots$ appear on this path in the order listed (possibly with repetition, and $\ldots (\mathcal{A}_2 \cup \mathcal{C}_2) \supseteq (\mathcal{A}_1 \cup \mathcal{C}_1)$.*

Proof. We shall use an inductive argument similar to the one used in the proof of Lemma 3.2. It is clear that forward_visit(e) and the recursive call preserve the property asserted in the lemma. After backing up a non-leftmost edge $u \to v$, the property still holds by induction because STACK is restored to its state just before the recursive call dfs_step(v). After backing up a leftmost edge, the property still holds because of the **while** loop on line 1 in Table 8.

Suppose now that during the search, we are at the end of dfs_step(u) and want to load list(u) on STACK. The following is an outline of a proof that load_triples(u) preserves the

required property of STACK.
line 10: Since c_1 is adjacent to a vertex in $\mathcal{B}_1$, this implies a B_2-A'_2 edge.
line 11: In this case, $A_2 \subseteq A_1$ because of the ordering of list(u), and $(A'_2 \cup D(q_2)) \subseteq A'_1$.
line 13: In this case, $A_2 \subseteq A_1$, $A'_2 \subseteq A'_1$, $A''_2 \subseteq A''_1$, $C_2 = C_1$ and $C'_2 \subseteq (A''_1 \cup C'_1)$.
line 15: Note, in this case, that $c_1 < c_2$ cannot happen. (If $a_1 = a_2$, this follows from the ordering of list(u). If $a_1 < a_2$, then it follows because otherwise, (a_1, b_1, c_1) would never have been generated.) Since c_1 is adjacent to a vertex in $\mathcal{B}_1$, this implies a B_2-C_2 edge.
lines 16–34: If $b_1 \neq b_2$, note that b_2 is a proper leftmost descendant of b_1 and that a_2 is a proper ancestor of b_1.
line 18: $c_1 \in B_2$, hence $HIGH1(c_1) > a_1$ implies a B_2-A_2 edge.
line 20: there is a B_2-A_2 edge.
line 22: Lemma 2.5 for (a_1, b_1, c_1) implies a B_2-A_2 edge.
line 25: Lemma 2.5 for (a_2, b_2, c_2) implies a B_1-A''_1 edge.
line 26: In this case, since $RCH(b_1) < \alpha_1$, it follows that there is a B_2-A''_2 edge.
line 28: Note that, in this case, c_1 and c_2 may be incomparable under the ancestor relation (if q_2 is non-leftmost) or c_2 is an ancestor of c_1. In the first case, $RCH(b_1) < \alpha_1$ implies a B_2-A'_2 edge. In the second case, it implies a B_2-A''_2 edge.
line 30: In this case, there is a B_2-A''_2 edge.
line 31: Since $\{a_1, c_2\}$ cannot be a vertex cut, it follows that there is a vertex v in the interior of the path $a_1 \rightarrow^* c_2$ such that either $RCH(v) < \alpha_2$ or there is a non-leftmost edge e in Adj(v) with $HIGH1(e) > a_1$. Now v cannot be in the interior of the paths $b_2 \rightarrow^* c_2$ or $b_1 \rightarrow^* c_1$ from Theorem 3.3 applied to (a_2, b_2, c_2) and (a_1, b_1, c_1) respectively. Also, it is clear that $v \neq b_1$. Hence v must be in the interior of the path $a_1 \rightarrow^* b_1$, and that implies a B_1-$\mathcal{A}_1$ edge.
line 33: Clearly, $A_2 \subseteq A_1$. If q_1 is non-leftmost, then $D(b_2) \subseteq A'_1$. Otherwise, if c_1 is an ancestor of b_2, then $(A'_2 \cup D(q_2)) \subseteq A''_1$. Otherwise, if $c_1 = c_2$, then $(A'_2 \cup C_2) \subseteq C_1$, $A''_2 \subseteq (A''_1 \cup C'_1)$ and $C'_2 \subseteq (A''_1 \cup C'_1)$.
line 34: If q_2 is non-leftmost, the fact that c_1 is adjacent to a vertex in $\mathcal{B}_1$ implies a B_2-A'_2 edge. Otherwise, if c_2 is a proper ancestor of c_1, it implies a B_2-A''_2 edge. Otherwise, c_1 is a proper ancestor of c_2 and the fact that c_1 is adjacent to a vertex in $\mathcal{B}_1$ implies a B_2-C_2 edge. □

Theorem 5.4 *The detection step correctly discards, in $O(n + m)$ time, those of its input triples that have a B-$(\mathcal{A} \cup \mathcal{C})$ edge.*

Proof. Suppose a triple (a, b, c) is discarded by the search for having a B-$(\mathcal{A} \cup \mathcal{C})$ edge. We need to verify that it indeed has one. If the triple is discarded by the subroutine load_triples, then it can be easily seen from the proof of the previous lemma that it has a B-$(\mathcal{A} \cup \mathcal{C})$ edge. If the triple is discarded by line 3 in Table 7, the edge $e = u \rightarrow v$ is such that the subtree $D(v) \subseteq B$ has an attachment higher than a, that is, in A. Hence the triple has a B-A edge. If the triple is marked by line 4 in Table 8, the edge $e = u \rightarrow v$ is non-leftmost, hence by Lemma 2.1 it follows that the triple has a B-A edge. Finally, if the triple is marked

by line 8 in Table 8, the vertex $u \in B$ has a back edge coming in from a vertex in $\mathcal{A} \cup \mathcal{C}$, hence the triple has a B-$(\mathcal{A} \cup \mathcal{C})$ edge.
Conversely, suppose a triple (a, b, c) has a B-$(\mathcal{A} \cup \mathcal{C})$ edge. Then it has either a B-$(A' \cup A'' \cup C \cup C')$ edge, or a B-A edge (or both.) From the previous lemma, it then follows that in the first case, such a triple would be discarded by the **while** loop on line 7 in Table 8. In the second case, it would be discarded by the **while** loop in Table 7 or line 4 in Table 8. Hence a triple with a B-$(\mathcal{A} \cup \mathcal{C})$ edge will be discarded.
Finally, for the time bound, note that the subroutine load_triples takes $O(1)$ time per triple, and hence loading list(u) for all vertices u takes $O(n+m)$ time overall (as there are $O(n+m)$ triples.) □

5.3 Detecting B-$(\mathcal{A} \cup \mathcal{C})$ edges in singular triples

This step is for non-degenerate triples with an $\mathcal{A}$-$\mathcal{C}$ edge. Before the detection step, the triples (a, b, c) such that c is not adjacent to any vertex in $\mathcal{B}$ are marked as having a B-$(\mathcal{A} \cup \mathcal{C})$ edge, and hence not examined during the detection step. (Refer to the corresponding argument in Section 5.2.)

The forbidden set for a triple is $\mathcal{A} \cup \mathcal{C}$ and the detection is carried out along the path $p \rightarrow^* b$. The subroutine sort_lists sorts list(u) $\forall u \in V$ such that (a_1, u, c_1) precedes (a_2, u, c_2) in the list iff $a_1 < a_2$, or $a_1 = a_2$ and $c_1 \leq c_2$. The pseudo-code for forward_visit(e) and backward_visit(e) is similar to that in tables 7 and 8 respectively. The difference is that instead of discarding a triple, we only mark it as having a $\mathcal{B}$-$(\mathcal{A} \cup \mathcal{C})$ edge and remove it from STACK. The pseudo-code for load_triples(u) is given in Table 10.

```
1  while list(u) is non-empty do begin
2      let (a1, b1, c1) be at the end of list(u)
3      if the current block of STACK is empty then
4          remove (a1, b1, c1) from list(u) and add it at the beginning of the current block
           of STACK
5      else begin
6          let (a2, b2, c2) be at the beginning of the current block of STACK
7          if b1 = b2 then
8              if q1 ≠ q2 then
9                  if HIGH1(q1) > a2 then
10                     mark (a2, b2, c2) as having a B-A edge and remove it from STACK
11                 else remove (a1, b1, c1) from list(u) and add it at the beginning of the
                   current block of STACK
12             else if c2 is a proper ancestor of c1 then
13                 if (A2 ∪ C2) ⊆ (A1 ∪ C1) then
14                     remove (a1, b1, c1) from list(u) and add it at the beginning of the
                       current block of STACK
15                 else mark (a2, b2, c2) as having a B-A edge and remove it from STACK
16             else if c1 = c2 then
```

```
17                  remove (a_1, b_1, c_1) from list(u) and add it at the beginning of the cur-
                    rent block of STACK
18              else mark (a_2, b_2, c_2) as having a B-C edge and remove it from STACK
19          else if a_1 > a_2 then
20              if c_1 ≥ b_2 OR c_1 ∉ D(q_2) then
21                  mark (a_2, b_2, c_2) as having a B-A edge and remove it from STACK
22              else if c_1 > c_2 then
23                  mark (a_2, b_2, c_2) as having a B-C edge and remove it from STACK
24              else mark (a_1, b_1, c_1) as having a B-C edge and remove it from list(u)
25          else if c_1 ≥ b_2 then
26              remove (a_1, b_1, c_1) from list(u) and add it at the beginning of the current
                block of STACK
27          else if c_1 ∉ D(q_2) then
28              if (A_2 ∪ C_2) ⊆ (A_1 ∪ C_1) then
29                  remove (a_1, b_1, c_1) from list(u) and add it at the beginning of the cur-
                    rent block of STACK
30              else mark (a_2, b_2, c_2) as having a B-A edge and remove it from STACK
31          else if c_2 is a proper ancestor of c_1 then
32              if (A_2 ∪ C_2) ⊆ (A_1 ∪ C_1) then
33                  remove (a_1, b_1, c_1) from list(u) and add it at the beginning of the cur-
                    rent block of STACK
34              else mark (a_2, b_2, c_2) as having a B-A edge and remove it from STACK
35          else if c_1 = c_2 then
36              remove (a_1, b_1, c_1) from list(u) and add it at the beginning of the current
                block of STACK
37          else mark (a_2, b_2, c_2) as having a B-C edge and remove it from STACK
38      end
39  end
```

Table 10: Detecting B-$(\mathcal{A} \cup \mathcal{C})$ edges in singular triples: pseudo-code for load_triples(u)

Lemma 5.5 *During the detection step, the following condition holds immediately before the* ***while*** *loops on line 2 of Table 7 and line 7 of Table 8. If u is the current vertex in the search, then the triples (a, b, c) in the current block of STACK are such that a and b are on the canonical path containing u, with a being a proper ancestor of u and b a (leftmost) descendant of u. Moreover, the triples $(a_1, b_1, c_1), (a_2, b_2, c_2), \ldots$ in the block (in order) will be such that the vertices $\ldots, a_2, a_1, u, b_1, b_2, \ldots$ appear on this path in the order listed (possibly with repetition), and $\ldots (\mathcal{A}_2 \cup \mathcal{C}_2) \supseteq (\mathcal{A}_1 \cup \mathcal{C}_1)$.*

Proof. We shall use an inductive argument similar to the one used in the proof of Lemma 3.2. It is clear that forward_visit(e) and the recursive call preserve the property asserted in the lemma. After backing up a non-leftmost edge $u \to v$, the property still holds by induction because STACK is restored to its state just before the recursive call dfs_step(v). After

backing up a leftmost edge, the property still holds because of the **while** loop on line 1 in Table 8.

Suppose now that during the search, we are at the end of dfs_step(u) and want to load list(u) on STACK. The following is an outline of a proof that load_triples(u) preserves the required property of STACK.
line 10: Since c_1 is adjacent to a vertex in $\mathcal{B}_1$, this implies a B_2-A'_2 edge.
line 11: Clearly, $A_2 \subseteq A_1$ and $A'_2 \subseteq A'_1$. Since $HIGH1(q_2) > a_2 \geq a_1$, $D(q_2) \subseteq A'_1$.
line 14: Clearly, (a_1, b_1, c_1) can be moved form list(u) to STACK.
line 15: Since $(\mathcal{A}_2 \cup \mathcal{C}_2) \not\subseteq (\mathcal{A}_1 \cup \mathcal{C}_1)$, $c_1 \in A''_2$. But since c_1 is adjacent to a vertex in $\mathcal{B}_1$, this implies a B_2-$\mathcal{A}_2$ edge.
line 17: Clearly, since $a_2 \geq a_1$, $A_2 \subseteq A_1$, $A'_2 \subseteq A'_1$, $A''_2 \subseteq A''_1$ and $C'_2 \subseteq (A''_1 \cup C'_1)$
line 18: If c_1 and c_2 are incomparable under the ancestor relation, then $c_1 \in C_2$ and since c_1 is adjacent to a vertex in $\mathcal{B}_1$, it follows that there is a B_2-C_2 edge. On the other hand, if c_1 is a proper ancestor of c_2, then $c_1 \in C_2$ and $a_1 < a_2$ (because of the ordering of list(u).) Let s be the child of c_1 with $c_2 \in D(s)$. Clearly, $D(s) \subseteq A''_1$, and since c_1 is adjacent to a vertex in $\mathcal{B}_1$, it follows that there is a B_2-C_2 edge.
lines 19–37: If $b_1 \neq b_2$, note that b_2 is a proper leftmost descendant of b_1 and that a_2 is a proper ancestor of b_1.
line 21: If q_1 is non-leftmost, then clearly there is a B_2-A_2 edge. If c_1 is a proper ancestor of b_2, let s be the child of c_1 with $b_2 \in D(s)$. Since a singular component of (a_2, b_2, c_2) has an attachment at a_2, $D(s)$ is not a singular component of (a_1, b_1, c_1) and it follows that there is a B_2-A_2 edge. Finally, if $c_1 = b_2$ or c_1 is a descendant of b_2 not contained in $D(q_2)$, a similar argument shows that there is a B_2-A'_2 edge.
line 23: It can be seen that c_2 is not contained in any singular component of (a_1, b_1, c_1). It follows that there is a B_2-C_2 edge.
line 24: It can be seen that c_1 is not contained in any singular component of (a_2, b_2, c_2). It follows that there is a $\mathcal{B}_1$-$\mathcal{C}_1$ edge.
line 26: If q_1 is non-leftmost, then $D(b_2) \subseteq A'_1$. If c_1 is an ancestor of b_2, then $(A'_2 \cup D(q_2)) \subseteq A''_1$. (Note that $HIGH1(q_2) > a_2 \geq a_1$ since the triples in this step are non-degenerate.)
line 29: Clearly, (a_1, b_1, c_1) can be moved from list(u) to STACK.
line 30: This happens only when $c_1 \in A'_2$, in which case, there is a B_2-A'_2 edge (as c_1 is adjacent to a vertex in $\mathcal{B}_1$.)
line 33: Clearly, (a_1, b_1, c_1) can be moved from list(u) to STACK.
line 34: In this case, $c_1 \in A''_2$, and there is a B_2-A''_2 edge (as c_1 is adjacent to a vertex in $\mathcal{B}_1$.)
line 36: In this case, $(A'_2 \cup C_2) \subseteq C_1$, $A''_2 \subseteq A''_1$ and $C'_2 \subseteq (A''_1 \cup C'_1)$
line 37: It can be seen that c_2 is not in any singular component of (a_1, b_1, c_1), hence there is a B_2-C_2 edge. □

Theorem 5.6 *The detection step correctly detects, in $O(n + m)$ time, those of its input triples that have a B-$(\mathcal{A} \cup \mathcal{C})$ edge.*

Proof. The proof of this is nearly identical to the proof of Theorem 5.4 and is hence omitted.

5.4 Detecting B-$\mathcal{A}$ edges

This step is for non-degenerate triples with no $\mathcal{A}$-$\mathcal{C}$ edges. The forbidden set for a triple is $\mathcal{A}$ and the detection is carried out on the path $p \rightarrow^* b$. The subroutine sort_lists sorts list(u) $\forall u \in V$ such that (a_1, u, c_1) precedes (a_2, u, c_2) in the list iff $a_1 < a_2$, or $a_1 = a_2$ and $c_1 \geq c_2$.

The pseudo-code for forward_visit(e) and backward_visit(e) is similar to that in tables 7 and 8 respectively. The difference is that instead of discarding a triple, we only mark it as having a B-$\mathcal{A}$ edge and remove it from STACK. Also, the forbidden set in this case is $\mathcal{A}$ instead of $\mathcal{A} \cup \mathcal{C}$ (see line 7 in Table 8.) The pseudo-code for load_triples(u) is given in Table 11.

Lemma 5.7 *During the detection step, the following condition holds immediately before the* ***while*** *loops on line 2 of Table 7 and line 7 of Table 8. If u is the current vertex in the search, then the triples (a, b, c) in the current block of STACK are such that a and b are on the canonical path containing u, with a being a proper ancestor of u and b a (leftmost) descendant of u. Moreover, the triples $(a_1, b_1, c_1), (a_2, b_2, c_2), \ldots$ in the block (in order) will be such that the vertices $\ldots, a_2, a_1, u, b_1, b_2, \ldots$ appear on this path in the order listed (possibly with repetition), and $\ldots \mathcal{A}_2 \supseteq \mathcal{A}_1$.*

Proof. We shall use an inductive argument similar to the one used in the proof of Lemma 3.2. It is clear that forward_visit(e) and the recursive call preserve the property asserted in the lemma. After backing up a non-leftmost edge $u \rightarrow v$, the property still holds by induction because STACK is restored to its state just before the recursive call dfs_step(v). After backing up a leftmost edge, the property still holds because of the **while** loop on line 1 in Table 8.

Suppose now that during the search, we are at the end of dfs_step(u) and want to load list(u) on STACK. The following is an outline of a proof that load_triples(u) preserves the required property of STACK. Note that for non-degenrate triples with no $\mathcal{A}$-$\mathcal{C}$ edges, an argument similar to the one in Lemma 2.3 shows that the path $q \rightarrow^* c$ is leftmost.
line 11: Clearly, there is a B_2-A'_2 edge.
line 12: Note that this happens only when $a_1 = a_2$ and $LOW1(q_1) = a_1$. Since c_1 is not adjacent to any vertex in $\mathcal{B}_1$, there is a $\mathcal{B}_1$-$(\mathcal{A}_1 \cup \mathcal{C}_1)$ edge. In fact, since $LOW1(q_1) = a_1$, there cannot be any $\mathcal{B}_1$-$\mathcal{C}_1$ edge, hence there must be a B_1-$\mathcal{A}_1$ edge.
line 13: Clearly, $A_2 \subseteq A_1$ and $(A'_2 \cup D(q_2)) \subseteq A'_1$.
line 14: Suppose $c_2 > c_1$. This means that $a_2 > a_1$ (because of the order of list(u)). It follows that c_1 is not in any singular component of (a_2, b_2, c_2), which implies an $\mathcal{A}_1$-$\mathcal{C}_1$ edge, a contradiction. Hence $c_1 \geq c_2$. It can now be seen that $A'_2 \subseteq A'_1$ and $A''_2 \subseteq A''_1$.
lines 15–22: If $b_1 \neq b_2$, note that b_2 is a proper leftmost descendant of b_1 and that a_2 is a proper ancestor of b_1.
line 17: If q_1 is non-leftmost, clearly there is a B_2-A_2 edge. If c_1 is a proper ancestor of b_2, it can be seen that b_2 is not contained in a singular component of (a_1, b_1, c_1), hence there is again a B_2-A_2 edge. If $c_1 = b_2$, $D(q_2)$ is not a singular component of (a_1, b_1, c_1). It follows

```
1  while list(u) is non-empty do begin
2        let (a_1, b_1, c_1) be at the end of list(u)
3        if the current block of STACK is empty then
4              remove (a_1, b_1, c_1) from list(u) and add it at the beginning of the current block
               of STACK
5        else begin
6              let (a_2, b_2, c_2) be at the beginning of the current block of STACK
7              if b_1 = b_2 then
8                    if q_1 ≠ q_2 then
9                          if HIGH1(q_1) > a_2 then
10                               if LOW1(q_1) < a_2 then
11                                     mark (a_2, b_2, c_2) as having a B-A edge and remove it from
                                       STACK
12                               else mark (a_1, b_1, c_1) as having a B-A edge and remove it from
                                 list(u)
13                         else remove (a_1, b_1, c_1) from list(u) and add it at the beginning of the
                           current block of STACK
14                   else remove (a_1, b_1, c_1) from list(u) and add it at the beginning of the
                     current block of STACK
15             else if a_1 > a_2 then
16                   if c_1 ≥ b_2 then
17                         mark (a_2, b_2, c_2) as having a B-A edge and remove it from STACK
18                   else    comment c_2 ≥ c_1
19                         mark (a_2, b_2, c_2) as having a B-A edge and remove it from STACK
20             else if c_1 ≥ b_2 then
21                   remove (a_1, b_1, c_1) from list(u) and add it at the beginning of the current
                     block of STACK
22             else mark (a_2, b_2, c_2) as having a B-A edge and remove it from STACK
23       end
24 end
```

Table 11: Detecting B-$\mathcal{A}$ edges: pseudo-code for load_triples(u)

that there is a B_2-A_2' edge.

line 19: First note that $c_1 > c_2$ is not possible. (In that case, c_1 would be a proper ancestor of c_2. Since c_2 is not contained in a singular component of (a_1, b_1, c_1), this would imply an $\mathcal{A}_2$-$\mathcal{C}_2$ edge, a contradiction.) Hence $c_2 \geq c_1$. If c_2 is an ancestor of c_1, it follows that a singular component of (a_1, b_1, c_1) implies a B_2-A_2'' edge. If not (i.e. if q_2 is non-leftmost) then a singular component of (a_1, b_1, c_1) implies a B_2-A_2' edge.

line 21: If q_1 is non-leftmost, then $D(b_2) \subseteq A_1'$. If c_1 is an ancestor of b_2, then $(A_2' \cup D(q_2)) \subseteq A_1''$. (Note that $HIGH1(q_2) > a_2 \geq a_1$ since the triples in this step are non-degenerate.)

line 22: If c_1 is an ancestor of c_2, then b_2 cannot be adjacent to any vertex in $\mathcal{A}_2$ (otherwise there would be an $\mathcal{A}_1$-$\mathcal{C}_1$ edge.) But then this means there is an $\mathcal{A}_2$-$(\mathcal{B}_2 \cup \mathcal{C}_2)$ edge, and

since there cannot be a $\mathcal{A}_2$-$\mathcal{C}_2$ edge, there is in fact a B_2-$\mathcal{A}_2$ edge. On the other hand, if $c_2 > c_1$, either c_2 is a proper ancestor of c_1, or q_2 is non-leftmost. In the first case, $c_1 \in A_2''$ (since (a_2, b_2, c_2) is non-degenerate) and there is a B_2-A_2'' edge. Similarly, in the latter case, $c_1 \in A_2'$ and there is a B_2-A_2' edge. □

Theorem 5.8 *The detection step correctly detects, in $O(n+m)$ time, those of its input triples that have a B-$\mathcal{A}$ edge.*

Proof. The proof of this is nearly identical to the proof of Theorem 5.4 and is hence omitted. □

5.5 Detecting C-$\mathcal{B}$ edges

This step is for non-degenerate triples with no $\mathcal{A}$-$\mathcal{C}$ edges, and also for degenerate triples, which, as observed before, do not have any $\mathcal{A}$-$\mathcal{C}$ edges either. The forbidden set for a triple is $\mathcal{A} \cup \mathcal{B}$ and the detection is carried out on the path $q \rightarrow^* c$. ($\mathcal{A}$ is included in the forbidden set only for convenience; it does not affect the detection step as it is only used for triples with no $\mathcal{A}$-$\mathcal{C}$ edges.) The subroutine sort_lists sorts list(u) $\forall u \in V$ such that (a_1, b_1, u) precedes (a_2, b_2, u) in the list iff $b_1 < b_2$, or $b_1 = b_2$ and $a_1 \leq a_2$.

The pseudo-code for replacing forward_visit(e) and backward_visit(e) is given in tables 12 and 13 respectively.

```
1  if e is non-leftmost then begin
2       while the triple (a,b,c) in the beginning of the current block of STACK has
        HIGH1(e) > b and HIGH1(e) ≠ a do
3            mark the triple as having a C-B edge and remove it from STACK
4       if e is a tree edge then add an end-marker on top of STACK
5  end
```

Table 12: Detecting C-$\mathcal{B}$ edges: pseudo-code for forward_visit(e)

Table 14 gives the pseudo-code for load_triples(u).

Lemma 5.9 *During the detection step, the following condition holds immediately before the **while** loops on line 2 of Table 12 and line 9 of Table 13. If u is the current vertex in the search, then the triples (a, b, c) in the current block of STACK are such that a,b and c are all on the canonical path containing u, with a and b being proper ancestors of u and c a (leftmost) descendant of u. Moreover, the triples $(a_1, b_1, c_1), (a_2, b_2, c_2), \ldots$ in the block (in order) will be such that the vertices $\ldots, b_2, b_1, u, c_1, c_2, \ldots$ appear on this path in the order listed (possibly with repetition), and $\ldots (\mathcal{A}_2 \cup \mathcal{B}_2) \supseteq (\mathcal{A}_1 \cup \mathcal{B}_1)$.*

```
1  while the triple (a, b, c) at the beginning of the current block of STACK has b = u do
2        remove the triple from STACK
3   if e = (u → v) is non-leftmost then begin    comment  backing up over a
                                                          non-leftmost edge
4        mark all triples in the current block of STACK as having a C-B edge
5        remove the block (and the end-marker) from STACK
6  end
7  else   comment backing up over a leftmost edge
8        for edge w ↪ u in Adj^R(u) do

9            while the triple (a, b, c) at the beginning of the current block has w ∈
             (A ∪ B) do
10               mark the triple as having a C-B edge
```

Table 13: Detecting C-$\mathcal{B}$ edges: pseudo-code for backward_visit(e)

Proof. We shall use an inductive argument similar to the one used in the proof of Lemma 3.2. It is clear that forward_visit(e) and the recursive call preserve the property asserted in the lemma. After backing up a non-leftmost edge $u \to v$, the property still holds by induction because STACK is restored to its state just before the recursive call dfs_step(v). After backing up a leftmost edge, the property still holds because of the **while** loop on line 1 in Table 13.

Suppose now that during the search, we are at the end of dfs_step(u) and want to load list(u) on STACK. The following is an outline of a proof that load_triples(u) preserves the required property of STACK.

line 9: Since $a_2 > a_1$, and there is no $\mathcal{A}_1$-$\mathcal{C}_1$ edge, a_2 cannot be adjacent to any vertex in $\mathcal{C}_2$. It follows that there must be a C_2-$\mathcal{B}_2$ edge.

line 11: If $a_1 \leq b_2$, then $(A_2 \cup B_2 \cup A_2' \cup B_2') \subseteq A_1$ and $(A_2'' \cup B_2'') \subseteq A_1''$. If $a_1 = a_2$, then $A_2 = A_1$, $(B_2 \cup A_2' \cup B_2') \subseteq B_1$, $A_2'' = A_1''$ and $B_2'' \subseteq B_1''$.

line 12: Since there is no $\mathcal{A}_2$-$\mathcal{C}_2$ edge, by considering a singular component of (a_1, b_1, c_1) it follows that $b_2 < a_1 < a_2$, and hence there is a C_2-B_2'' edge.

lines 13–21: If $c_1 \neq c_2$, note that c_2 is a proper leftmost descendant of c_1 and that b_2 is a proper ancestor of c_1.

line 14: Note that c_2 is not contained in any singular component of (a_1, b_1, c_1). Now a singular component of (a_1, b_1, c_1) must have an attachment other than b_2 and a_2, and since there is no $\mathcal{A}_2$-$\mathcal{C}_2$ edge, it follows that there must be a C_2-B_2 edge.

line 17: If $a_1 \leq b_2$, then $(A_2 \cup B_2 \cup A_2' \cup B_2') \subseteq A_1$ and $(A_2'' \cup B_2'') \subseteq D(c_2) \subseteq A_1''$. (Note that this case cannot happen for degenerate triples.) If $a_1 = a_2$, then $A_2 = A_1$ and $(B_2 \cup A_2' \cup B_2') \subseteq B_1$. Furthermore, $(A_2'' \cup B_2'') \subseteq A_1''$ in the case of non-degenerate triples, whereas $(A_2'' \cup B_2'') \subseteq B_1''$ in the case of degenerate triples.

line 18: The argument is similar to that for line 12.

```
1  while list(u) is non-empty do begin
2        let (a_1, b_1, c_1) be at the end of list(u)
3        if the current block of STACK is empty then
4              remove (a_1, b_1, c_1) from list(u) and add it at the beginning of the current block
               of STACK
5        else begin
6              let (a_2, b_2, c_2) be at the beginning of the current block of STACK
7              if c_1 = c_2 then
8                    if b_1 = b_2 then
9                          mark (a_2, b_2, c_2) as having a C-B edge and remove it from STACK
10                   else if a_1 ≤ b_2 OR a_1 = a_2 then
11                         remove (a_1, b_1, c_1) from list(u) and add it at the beginning of the cur-
                           rent block of STACK
12                   else mark (a_2, b_2, c_2) as having a C-B edge and remove it from STACK
13             else if b_1 > b_2 then
14                   mark (a_2, b_2, c_2) as having a C-B edge
15             else if b_1 < b_2 then
16                   if a_1 ≤ b_2 OR a_1 = a_2 then
17                         remove (a_1, b_1, c_1) from list(u) and add it at the beginning of the cur-
                           rent block of STACK
18                   else mark (a_2, b_2, c_2) as having a C-B edge and remove it from STACK
19             else if a_1 = a_2 then
20                   remove (a_1, b_1, c_1) from list(u) and add it at the beginning of the current
                     block of STACK
21             else mark (a_2, b_2, c_2) as having a C-B edge
22       end
23 end
```

Table 14: Detecting C-$\mathcal{B}$ edges: pseudo-code for load_triples(u)

line 20: In this case, A_2, B_2, A_2' and B_2' are respectively identical to A_1, B_1, A_1' and B_1'. In the case of non-degenerate triples, $(A_2'' \cup B_2'') \subseteq A_1''$. In the case of degenerate triples, $(A_2'' \cup B_2'') \subseteq B_1''$.
line 21: In this case, $a_1 < a_2$, since there is no $\mathcal{A}_2$-$\mathcal{C}_2$ edge. Now a singular component of (a_1, b_1, c_1) gives a C_2-B_2 edge. □

Theorem 5.10 *The detection step correctly marks, in $O(n+m)$ time, those of its input triples that have a C-$\mathcal{B}$ edge.*

Proof. Suppose a triple (a, b, c) is marked by the search as having a C-$\mathcal{B}$ edge. We need to verify that it indeed has one. If the triple is marked by the subroutine load_triples, then it can be easily seen from the proof of the previous lemma that it has a C-$\mathcal{B}$ edge. If the triple is marked by line 3 in Table 12, the edge $e = u \to v$ is such that the subtree $D(v) \subseteq C$ has an

attachment distinct from a and higher than b, that is, in B (since the input triples for this step have no $\mathcal{A}$-$\mathcal{C}$ edges.) Hence the triple has a C-B edge. If the triple is marked by line 4 in Table 13, the edge $e = u \rightarrow v$ is non-leftmost, hence by Lemma 2.3 it follows that the triple has $HIGH1(q) \leq a$ (i.e. the input triples are degenerate.) It follows from the ordering of Adj(u) that the first edge e_0 in Adj(u) (or any edge before e) must have $HIGH1(e_0) = a$ and hence be a tree edge. Now if $HIGH2(e_0) \leq b$, it would mean $HIGH2(e) \leq b$, and hence c cannot be adjacent to any vertex in $\mathcal{B}$. Since the triple is degenerate, this would mean that it has a C-$\mathcal{B}$ edge. On the other hand, if $HIGH2(e_0) > b$, and $e_0 = u \rightarrow v_0$, say, then the subtree $D(v_0) \subseteq C$ has an attachment in B, so the triple has a C-B edge. Finally, if the triple is marked by line 10 in Table 13, the vertex $u \in C$ has a back edge coming in from a vertex in B'', hence the triple has an C-B'' edge.
Conversely, suppose a triple (a, b, c) has a C-$\mathcal{B}$ edge. Then it has either a C-B'' edge, or a C-B edge (or both.) From the previous lemma, it then follows that in the first case, such an edge would be detected by the **while** loop on line 9 in Table 13. In the second case, the edge would be detected by the **while** loop in Table 12 or line 4 in Table 13. Hence a triple with a C-$\mathcal{B}$ edge will be marked accordingly.
Finally, for the time bound, note that the subroutine load_triples takes $O(1)$ time per triple, and hence loading list(u) for all vertices u takes $O(n+m)$ time overall (as there are $O(n+m)$ triples.) □

5.6 Detecting C-B'' edges

This step is carried out for non-singular triples that do not have any B-$(\mathcal{A} \cup \mathcal{C})$ edges, and for non-degenerate triples that have an $\mathcal{A}$-$\mathcal{C}$ edge but no $\mathcal{B}$-$(\mathcal{A} \cup \mathcal{C})$ edges. For all vertices u, list(u) is the list of triples, as above, with $c = u$, sorted such that (a_1, b_1, u) precedes (a_2, b_2, u) in the list iff $a_1 > a_2$, or $a_1 = a_2$ and $b_1 \leq b_2$.

For each vertex u, we first divide up list(u) into ordered *clusters* and the tree edges in Adj(u) into *sublists* corresponding to the clusters as follows. We scan list(u) in order and form clusters corresponding to (contiguous) subsequences of triples with non-decreasing values of b. It can be seen that the triples $(a_1, b_1, u), (a_2, b_2, u), \ldots, (a_k, b_k, u)$ in a cluster (in order) are such that the vertices $a_1, \ldots, a_k$ and $b_1, \ldots, b_k$ appear in the order $a_1, a_2, \ldots, a_k, b_k, \ldots, b_2, b_1$ (possibly with repetition) on the path from the root to u. We then scan Adj(u) in its usual order and make sublists of the *tree* edges in it, each sublist corresponding to a cluster in such a way that an edge e appears in the sublist corresponding to a cluster as above if $b_1 < HIGH1(e) \leq a_1$. Note that an edge can appear in at most one such sublist because the clusters do not "overlap" i.e. if we have a cluster $(a_1, b_1, u), (a_2, b_2, u), \ldots$ as above, and a cluster $(a'_1, b'_1, u), (a'_2, b'_2, u), \ldots$, then a_1,a'_1,b_1 and b'_1 cannot appear in that order (without repetition) on the path from the root to u. For non-singular triples, this is not possible because then Lemma 2.6 would imply that (a'_1, b'_1, u) has a B-$\mathcal{A}$ edge. Similarly, for non-degenerate triples, this is not possible because a singular component of (a_1, b_1, u) would then give rise to a B-$\mathcal{A}$ edge for (a'_1, b'_1, u).

We then sort all the sublists of Adj(u) for all vertices u in increasing order of $LOW1$.

Note that for this sorting to be done in linear time overall, we use bucket sort to sort all the sublists (for all vertices u) *simultaneously*. The pseudo-code for detecting C-B'' edges is given in Table 15.

```
1  for each vertex u do
2      for each cluster of list(u) do
3          let e be the first edge in the sublist of Adj(u) corresponding to the cluster and
           (a, b, u) be the first triple in the cluster
4          while the end of the cluster or the end of the sublist is reached do
5              if LOW1(e) ≥ b then
6                  set (a, b, u) to the next triple in the cluster
7              else if b < HIGH1(e) < a OR (HIGH1(e) = a AND b < HIGH2(e) < a) then
8                  mark the triple as having a C-B'' edge and set (a, b, u) to the next
                   triple in the cluster
9              else set e to the next edge in the sublist
```

Table 15: Detecting C-B'' edges

Theorem 5.11 *The procedure in Table 15 detects, in $O(m+n)$ time, the triples that have a C-B'' edge.*

Proof. For a particular cluster and its corresponding sublist, the **while** loop takes as many steps as the number of triples in the cluster plus the number of edges in the sublist (plus a constant.) Since the clusters and sublists are disjoint, this implies that for a vertex u, the time taken by the inner **for** loop is linear in the size of list(u) plus the degree of u. Since there are $O(m+n)$ triples, the linear time bound follows.

Consider a triple (a, b, u) in list(u). If it is marked by the procedure as having a C-B'' edge, then it follows from the condition in line 7 of Table 15 that the triple indeed has a C-B'' edge. On the other hand, suppose the triple has such an edge. We need to show that it is marked so by the procedure. Let $e = (u \rightarrow v)$ be the first edge in the sublist corresponding to the cluster containing the triple such that $D(v) \in B''$ and $D(v)$ has an attachment in C. Then either $b < HIGH1(e) < a$, or $HIGH1(e) = a$ and $b < HIGH2(e) < a$. Since the sublist is ordered in increasing order of $LOW1$, the triple is still being considered (or waiting to be considered) when the edge e is being examined by the procedure. Furthermore, the triple is then marked before the procedure moves to the next edge in the sublist. This is because, for a triple occuring before (a, b, u) in the cluster, one of the conditions on lines 5 or 7 must hold, and hence the last option in the **if** statement (line 9) cannot happen. In other words, before the procedure moves to the next edge (after e) in the sublist, the triple (a, b, u) will be considered, and hence will be marked as having a C-B'' edge. This proves the correctness of the procedure, and hence proves the theorem. □

6 Conclusion

We have presented a linear time algorithm to find the 3-shredders in a 3-connected graph. The analogous problem for 4-shredders in "internally 4-connected graphs" will be of great interest, because of its application to the even directed cycle problem and a host of equivalent problems mentioned in the introduction. We hope that a linear time algorithm for the even directed cycle problem can be obtained by further elaboration on the techniques presented in this paper.

References

[BS 95] Brualdi, R.A. and Shader, B.L., Matrices of sign-solvable linear systems, *C*ambridge Tracts in Mathematics **116** (1995).

[CT 98] Cheriyan, J. and Thurimella, R., "Fast Algorithms for k-Shredders and k-Node Connectivity Augmentation", *J.* Algorithms **33** (1999), No.1, 15–50.

[GT 85] Gabow, Harold N. and Tarjan, Robert Endre, A linear-time algorithm for a special case of disjoint set union, *J.* Comput. System Sci. **30** (1985), no. 2, 209–221.

[J 98] Gustedt, Jens, Efficient Union-Find for planar graphs and other sparse graph classes, *T*heoret. Comput. Sci. **203** (1998), no. 1, 123–141.

[GM 00] Gutwenger, C. and Mutzel, P., "A Linear Time Implementation of SPQR-Trees", *P*roc. Graph Drawing 2000, Lecture Notes in Computer Science Vol. 1984, 77–90.

[HRG 96] Henzinger, M. R., Rao, S. and Gabow, H.N., "Computing Vertex Connectivity: New Bounds from Old Techniques", *P*roc. 37th IEEE F.O.C.S. (1996), 462–471.

[HT 73] Hopcroft, J. E. and Tarjan, R. E., "Dividing a Graph into Triconnected Components", *S*IAM J. Comput. **2** (1973), No. 3, 135–158.

[Kas 63] Kastelyn, P.W., Dimer Statistics and Phase Transitions, *J.* Math. Phys. **4** (1963), 287–293.

[Kas 67] Kastelyn, P.W., Graph Theory and Crystal Physics, *G*raph Theory and Theoretical Physics (F. Harary, ed.), Academic Press, New York (1967), 43–110.

[KLM 84] Klee, V., Ladner, R. and Manber, R., Sign-Solvability revisited, *L*inear Algebra Appl. **59** (1984), 131–158.

[MRST 97] McCuaig, W., Robertson, N., Seymour, P.D. and Thomas, R., Permanents, Pfaffian Orientations, and Even Directed Circuits (Extended abstract), *P*roc. STOC (1997).

[Pól 13] Pólya, G., Aufgabe 424, *A*rch. Math. Phys. Ser. **20** (1913), 271.

[RST 99] Robertson, N., Seymour, P.D. and Thomas, R., Permanents, Pfaffian Orientations, and Even Directed Circuits, *A*nn. Math. **150** (1999), 929–975.

[Sey 80] Seymour, P.D., Disjoint paths in graphs, *D*iscrete Math. **29**, (1980), 293–309.

[Sh 80] Shiloach Y., A polynomial solution to the undirected two paths problem, *J.* Assoc. Comput. Mach. **27** (1980), 445–456.

[T 75] Tarjan, Robert Endre, Efficiency of a good but not linear set union algorithm, *J.* Assoc. Comput. Mach. **22** (1975), 215–225.

[Th 80] Thomassen C., 2 linked graphs, *E*ur. J. Comb. **1** (1980), 371–378.

[VY 89] Vazirani, V.V. and Yannakakis, M., Pfaffian orientations, 0-1 permanents, and even cycles in directed graphs, *D*iscrete Appl. Math. **25** (1989), 179–190.