Path-Based Depth-first Search for Strong and Biconnected Components*

Harold N. Gabow †

February 22, 2000

Abstract

Previous linear-time depth-first search algorithms for strong and biconnected components either compute auxiliary quantities based on the depth-first search tree (e.g., LOWPOINT values) or require two passes. We present one pass algorithms that only maintain a representation of the depth-first search path. This gives a simplified view of depth-first search without sacrificing efficiency.

1 Introduction

Depth-first search, as developed by Tarjan and co-authors, is a fundamental technique of efficient algorithm design for graphs [25]. This note presents depth-first search algorithms for two basic problems, strong and biconnected components. The algorithms are not based on the depth-first search tree but rather on the depth-first search path. This simplified view should make the algorithms easier to understand and code. The algorithms appear to be just as efficient as traditional depth-first search algorithms.

Most depth-first search algorithms (e.g., [25, 12, 13]) compute so-called LOWPOINT values that are defined in terms of the depth-first search tree. Because of the success of this method LOWPOINT values have become almost synonymous with depth-first search. LOWPOINT values are regarded as crucial in the strong and biconnected component algorithms, e.g. [16, pp. 94, 514]. Tarjan's LOWPOINT method for strong components is presented in texts [1, 9, 16, 18, 19, 23]. The strong component algorithm of Kosaraju and Sharir [24] is often viewed as conceptually simpler but it requires two passes over the graph. It is presented in texts [2, 5, 7, 28]. Tarjan's LOWPOINT biconnected component algorithm is presented in texts [1, 2, 5, 6, 9, 15, 16, 18, 19, 23, 28]. A two-pass biconnected component algorithm of Micali that avoids LOWPOINT values is sketched in [9, pp.67-68].

This paper presents strong and biconnected component algorithms that are based on the depth-first search path. This natural approach appears to have first been proposed by Purdom [21] and Munro [20] for strong components. It is regarded as requiring an extra data structure for set

*Technical Report CU-CS-890-99, revised version, Department of Computer Science, University of Colorado at Boulder.

†Department of Computer Science, University of Colorado at Boulder, Boulder, CO 80309. e-mail: hal@cs.colorado.edu

merging in order to be asymptotically efficient, and hence unlikely to be efficient in practice [25]. We present linear-time implementations of this approach for both strong and biconnected components. Our implementations use only stacks and arrays as data structures. A line-by-line pseudocode comparison of our algorithms with the tree-based algorithms of [25] shows the two approaches are similar in terms of lower level resource usage; performance differences are likely to be small or platform-dependent. Our algorithms show that the simpler path-based view of depth-first search suffices for these properties.

Our biconnected component algorithm can be used to find the bridges and articulation points of an undirected graph. If one is only interested in bridges there is a simpler strategy, which we present in an appendix. This algorithm also gives an immediate proof of Robbins' Theorem [22]. If one is only interested in articulation points we give another simple strategy in an appendix. One can design other path-based depth-first search algorithms for properties such as ear decomposition [17], *st*-numbering [9], topological numbering, etc.

After the initial writing of this paper Bob Tarjan communicated to us a strong component algorithm that can be interpreted as path-based. The algorithm is believed to date from the early 80's [27]. It is slightly more space efficient than the algorithm of Section 2. We sketch this algorithm in an appendix. We also give a variant of the algorithm of Section 2 that achieves the same space savings.

Section 2 presents our strong component algorithm and Section 3 presents the biconnected component algorithm. Appendix A presents the bridge algorithm. Appendix B proves a simple property of biconnected components. Appendix C presents the articulation points algorithm. Appendix D discusses the more space efficient algorithms. (Publication [10] is the same as this Technical Report except that it omits Appendices A,C and D because of space limitations.) We conclude this section with some terminology.

Singleton sets are usually denoted by omitting set braces, e.g., for a set S and element x, $S - x$ denotes $S - \{x\}$. We assume all input graphs contain n vertices and m edges.

We use the following operations to manipulate a stack S: $\texttt{PUSH}(x, S)$ adds x to S at the (new) top of S. $\texttt{POP}(S)$ removes the value at the top of the stack and returns that value. $\texttt{TOP}(S)$ is the index of the value at the top of the stack. Hence $S[\texttt{TOP}(S)]$ is the value at the top of the stack.

2 Strong Components

Consider a digraph $G = (V, E)$. Two vertices are in the same *strong component* of G if and only if they are mutually reachable, i.e., there is a path from each vertex to the other. The *strong component graph* is formed by contracting the vertices of each strong component. Equivalently the strong component graph is the acyclic digraph, formed by contracting vertices of G, that has as many vertices as possible. In short we say the strong component graph is the finest acyclic contraction of G.

This characterization suggests the following high-level algorithm to find the strong component graph of $G = (V, E)$. See Fig. 1. The algorithm maintains a graph H that is a contraction of G with some vertices deleted. It also maintains a path P in H. Initially H is the given graph G.

If H has no vertices stop. Otherwise start a new path P by choosing a vertex v and setting $P = (v)$. Continue by growing P as follows.

To grow the path $P = (v_1, \ldots, v_k)$ choose an edge (v_k, w) directed from the last vertex of P and do the following:

If $w \notin P$, add w to P, making it the new last vertex of P. Continue growing P.

If $w \in P$, say $w = v_i$, contract the cycle $v_i, v_{i+1}, \ldots, v_k$, both in H and in P. P is now a path in the new graph H. Continue growing P.

If no edge leaves v_k, output v_k as a vertex of the strong component graph. Delete v_k from both H and P. If P is now nonempty continue growing P. Otherwise try to start a new path P.

It is easy to see that this algorithm forms the finest acyclic contraction of G. (For instance if no edge leaves v_k then v_k is a vertex of the finest acyclic contraction.) Thus the algorithm correctly computes the strong components.

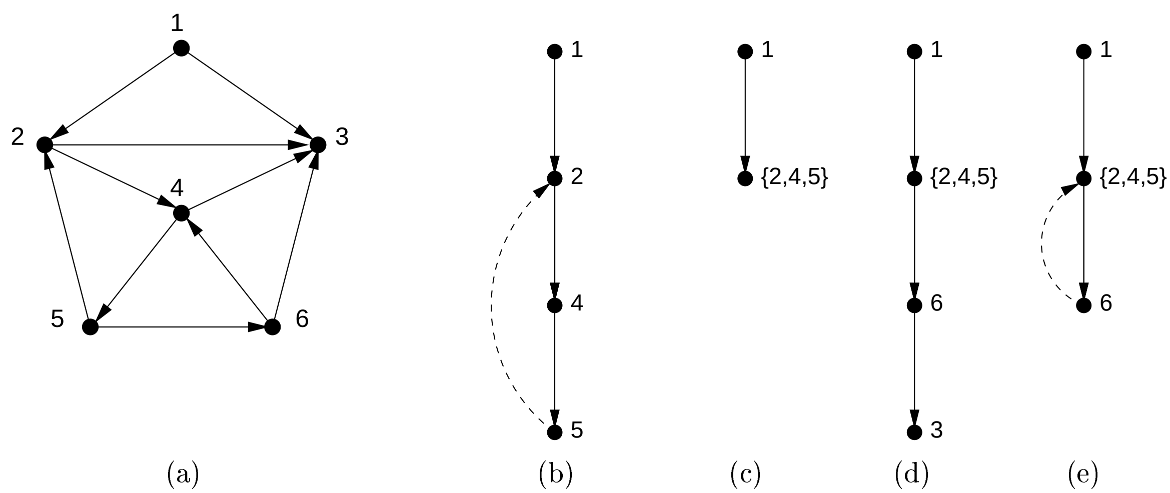
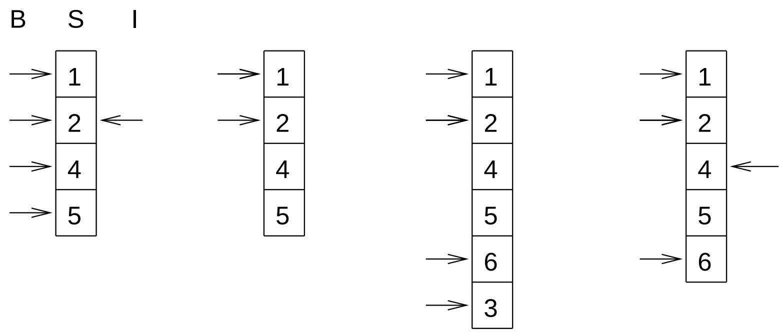


Figure 1: (a) Digraph G. (b)–(e) Path P (solid edges) in the first several steps of the algorithm. Strong component $\{3\}$ is output in (d).

This high-level algorithm was originally proposed by Purdom [21] and Munro [20]. The time for an efficient implementation is dominated by the time to keep track of the new vertices formed by contraction operations. Any data structure for disjoint set merging [7] can be used for this purpose. [21] and [20] use simple set-merging data structures, achieving total time $O(n^2)$ and $O(m + n \log n)$ respectively. Tarjan has shown set merging can be more efficiently, giving total time $O(m\alpha(m, n) + n)$ [26]. In fact the incremental tree set merging algorithm of [11] can be used. This reduces the time to $O(m + n)$, giving a linear time algorithm to find strong components. However the overhead of using incremental tree set merging may be significant in practice. Also the incremental tree algorithm requires a RAM machine and does not apply to a pointer machine. Now we give a simple list-based implementation that achieves linear time. The data structure is illustrated in Fig. 2.



(a) (b) (c) (d)

Figure 2: (a)–(d) show the data structure for Fig. 1(b)–(e) respectively. Stack S is shown. Arrows to the left of S represent stack B. Arrows to the right of S represent the entries of I that are used in contract steps. E.g. in (a) the algorithm reads $I[2] = 2$ and then contracts cycle $2, 4, 5$ to get (b). In (c) $I[3]$ changes from 6 to 7, the latter being the strong component number of vertex 3.

Assume the vertices of the given graph G are numbered by consecutive integers from 1 to n. The algorithm numbers the strong components of G by consecutive integers starting at $n + 1$. It records the strong component number for each vertex (see (2) below).

Two stacks are used to represent the path P. Stack S contains the sequence of (original) vertices in P and stack B contains the boundaries between contracted vertices. More specifically S and B correspond to $P = (v_1, \ldots, v_k)$ where $k = \texttt{TOP}(B)$ and for $i = 1, \ldots, k$,

(1) $$v_i = \{S[j] : B[i] \le j < B[i + 1]\}.$$

When $k > 0$ we have $B[1] = 1$. Also when $i = k$ in (1) we interpret $B[k + 1]$ to be $\texttt{TOP}(S) + 1$.

An array $I[1..n]$ is used to store stack indices. It also stores the strong component number of a vertex when that number is known. More precisely for a given vertex v at any point in time,

(2) $$I[v] = \begin{cases} 0 & \text{if } v \text{ has never been in } P; \\ j & \text{if } v \text{ is currently in } P \text{ and } S[j] = v; \\ c & \text{if the strong component containing } v \text{ has been deleted and numbered as } c. \end{cases}$$

Since there are only n vertices, there can be no confusion between an index j and a component number c in (2). A variable c is used to keep track of the component numbers.

The algorithm consists of a main routine `STRONG` and a recursive procedure `DFS`:

procedure $\texttt{STRONG}(G)$

1. empty stacks S and B;
2. **for** $v \in V$ **do** $I[v] = 0$;
3. $c = n$;
4. **for** $v \in V$ **do if** $I[v] = 0$ **then** $\texttt{DFS}(v)$;

procedure DFS(v)

1. PUSH(v, S); $I[v] =$ TOP(S); PUSH$(I[v], B)$; /* add v to the end of P */
2. **for** edges $(v, w) \in E$ **do**
3.   **if** $I[w] = 0$ **then** DFS(w)
4.   **else** /* contract if necessary */ **while** $I[w] < B[$TOP$(B)]$ **do** POP(B);
5. **if** $I[v] = B[$TOP$(B)]$ **then** { /* number vertices of the next strong component */
6.   POP(B); increase c by 1;
7.   **while** $I[v] \leq$ TOP(S) **do** $I[$POP$(S)] = c$ };

Theorem 2.1 *When* STRONG(G) *halts each vertex* $v \in V$ *belongs to the strong component numbered* $I[v]$*. The time and space are both* $O(m + n)$*.*

Proof: We will prove the first assertion of the theorem by showing that STRONG is a valid implementation of the high-level algorithm. We begin by specifying how the high-level algorithm will choose the edge (v_k, w) to grow P. Say that a vertex $w \in V$ becomes *active* (alternatively, *gets activated*) when it gets added to P as the new last vertex. The *most active* vertex is the currently active vertex that was activated most recently. To choose the next edge (v_k, w) let v be the most active vertex. Choose a previously unchosen edge directed from v, and use the corresponding edge of H as (v_k, w). If all edges directed from v have been chosen then deactivate v. If this makes all vertices of v_k inactive then output v_k as the next strong component.

We must verify that this strategy correctly implements the high-level algorithm. This is easily done by verifying that v is a vertex of v_k, i.e., the most active vertex always belongs to the last vertex of P.

Now we prove that STRONG implements this version of the high-level algorithm. We assume that H, P and the deleted strong components are as specified by (1)–(2). The argument is by induction on the number of statements executed in STRONG. We will mention some points about the various statements and leave the remaining straightforward details of the induction to the reader. We refer to lines of pseudocode by the initial of the procedure name followed by the line number e.g., D7 is the last line of DFS.

When S4 is being executed, P is empty. (By convention the execution of a line or a statement excludes the execution of any recursive call.) During the execution of the loop of D2, v is the most active vertex.

In D3 if $0 < I[w] \leq n$ then D4 contracts a cycle or does nothing if (v, w) has already been contracted. If $I[w] > n$ then the component containing w has been deleted and D4 does nothing.

We turn to showing that the time and space are $O(m + n)$. We assume the given graph G is stored as a collection of adjacency lists. Observe that every vertex is pushed onto and popped from each stack S, B exactly once. Hence it is easy to see that the algorithm spends $O(1)$ time on each vertex or edge. $\square$

Comparing our code to the algorithm of [25], both methods use stack S; our size n array I corresponds to a similar array that holds depth-first discovery numbers; our stack B corresponds to a size n array that holds LOWLINK values. Both S and B contain at most n entries at any time.

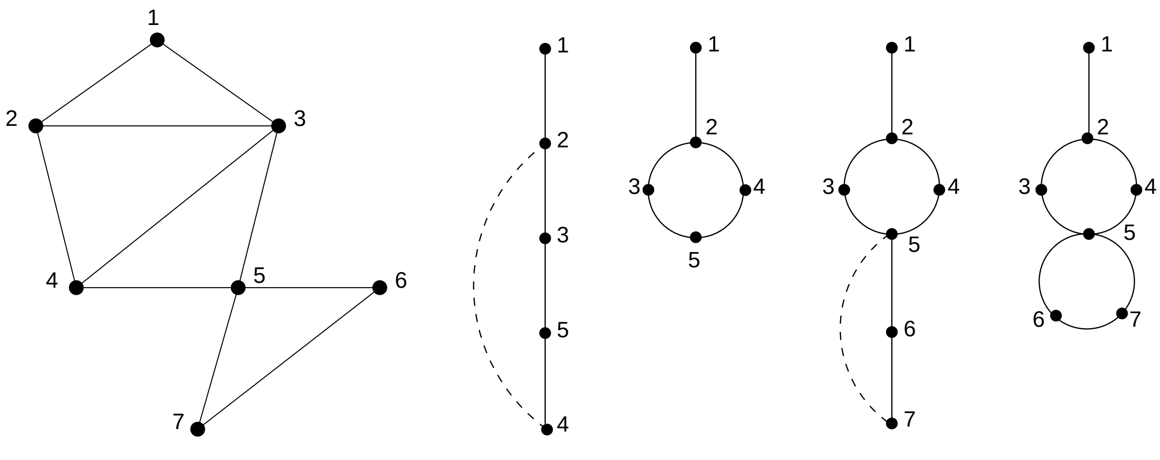


Figure 3: (a) Undirected graph G. (b)–(e) Path P (solid edges) in the first several steps of the algorithm. Biconnected component $\{5, 6, 7\}$ is output in (e).

3 Biconnected Components

We present our algorithm for biconnected components in the language of hypergraphs. This is not logically necessary but it brings out the similarity to the strong components algorithm.

We start by reviewing basic definitions about hypergraphs [4, 17]. A *hypergraph* $H = (V, E)$ consists of a finite set V of *vertices* and a finite set E of *edges*, each edge a subset of V. A *path* is a sequence $(v_1, e_1, \ldots, v_k, e_k)$ of distinct vertices v_i and distinct edges e_i, $1 \le i \le k$, with $v_1 \in e_1$ and $v_i \in e_{i-1} \cap e_i$ for every $1 < i \le k$. The set of all vertices in edges of P is denoted $V(P) = \bigcup_{i=1}^{k} e_i$. A *cycle* is a path with the additional properties that $k > 1$ and $v_1 \in e_k$. A hypergraph is *acyclic* if it contains no cycle. (This differs from the definition used in the study of databases [3].)

Notice that in a path P each vertex v_{i+1}, $1 \le i < k$ belongs to $e_i - v_i$. For this reason the sets $e_i - v_i$ figure prominently in our algorithm (e.g. see (4) below). The algorithm also uses this operation on hypergraphs: To *merge* a collection of edges e_i, $i = 1, \ldots, k$, add a new edge $\bigcup_{i=1}^{k} e_i$ and delete every edge of E contained in it (e.g., e_i). A *merging* of hypergraph H is a hypergraph formed by doing zero or more merges on H.

Now consider an undirected graph $G = (V, E)$. Two distinct edges are in the same *biconnected component* of G if and only if some simple cycle contains both of them. This relation is easily seen to be an equivalence relation over the edges, so the biconnected components are well-defined. The "block-cutpoint tree" of a graph represents the biconnected components and cutpoints [14]. We will use a hypergraph variant of this notion: The *block hypergraph* H of G is the hypergraph formed by merging the edges of each biconnected component of G. H is an acyclic hypergraph. In fact H can be characterized as the finest acyclic merging of G, i.e., it is the acyclic hypergraph formed by merging edges of G that has as many hyperedges as possible. For completeness this characterization is proved in Appendix B.

The characterization suggests the following high-level algorithm to find the block hypergraph of $G = (V, E)$. See Fig. 3. The algorithm maintains a hypergraph H that is a merging of G with

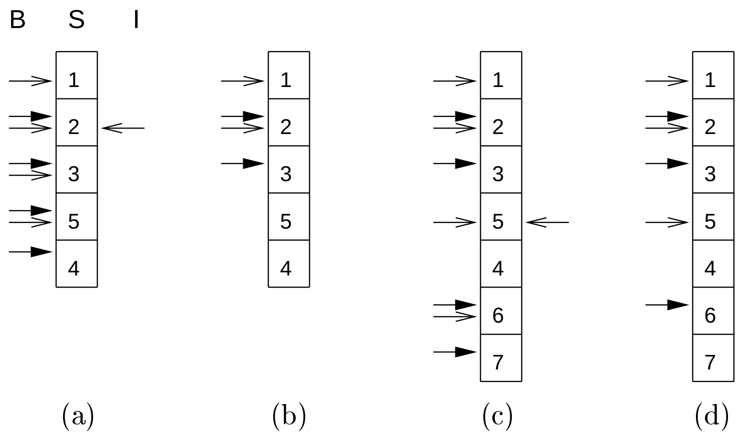


Figure 4: (a)–(d) illustrate the data structure for Fig. 3(b)–(e) respectively. S, B and I are represented as in Fig. 2. Every other arrowhead of B is drawn filled. E.g. in (a) the algorithm reads $I[2] = 2$ and then merges cycle $2, 3, 5, 4$ to get (b). In (d) $I[6]$ and $I[7]$ change to 8, the number of the first biconnected component.

some edges deleted, and a path P in H. Initially H is the given graph G.

If H has no edges stop. Otherwise start a new path P by choosing an edge $\{v, w\}$ and setting $P = (v, \{v, w\})$ (choose the end v arbitrarily). Continue by growing P as follows.

To grow the path $P = (v_1, e_1, \ldots, v_k, e_k)$ choose an edge $\{v, w\} \neq e_k$ with $v \in e_k - v_k$ and do the following:

If $w \notin V(P)$, add $v, \{v, w\}$ to the end of P. Continue growing P.

If $w \in V(P)$, say $w \in e_i - v_{i+1}$, merge the edges of the cycle $w, e_i, v_{i+1}, e_{i+1}, \ldots, v_k, e_k, v, \{v, w\}$ to a new edge $e = \bigcup_{j=i}^{k} e_j$, both in H and in P. P is now a path ending with e (i.e., (v_i, e) has replaced $(v_i, e_i, \ldots, v_k, e_k)$). Continue growing P.

If no edge leaves $e_k - v_k$, output e_k as an edge of the block hypergraph. Delete e_k from H and delete (v_k, e_k) from P. If P is now nonempty continue growing P. Otherwise try to start a new path P.

Correctness of this algorithm is based on two simple observations: When $v, \{v, w\}$ is added to P the result is a valid path, by the condition $v \in e_k - v_k$. When edges are merged they form a valid cycle, by the condition $\{v, w\} \neq e_k$. Now a straightforward inductive argument proves the algorithm correctly forms the finest acyclic merging of G, i.e., it finds the block hypergraph as desired.

As in Section 2 we give a list-based implementation that achieves linear time. The data structure is illustrated in Fig. 4. As before assume the vertices of G are numbered by consecutive integers from 1 to n. The algorithm numbers the biconnected components of G by consecutive integers starting at $n + 1$. The biconnected components are represented by assigning a number $I[v]$ to each vertex v in such a way that each edge $\{v, w\}$ belongs to the biconnected component with number $\min\{I[v], I[w]\}$ (see (5) below).

Two stacks are used to represent the path P. Stack S contains the vertices $V(P)$ and stack B represents the boundaries between edges of P, two vertices per boundary. More specifically S and

B correspond to $P = (v_1, e_1, \ldots, v_k, e_k)$, where $\mathtt{TOP}(B) = 2k$ and for $i = 1, \ldots, k$,

$$\begin{aligned} &(3) & v_i &= S[B[2i-1]]; \\ &(4) & e_i - v_i &= \{S[j] : B[2i] \le j < B[2i+2]\}. \end{aligned}$$

Thus in Fig. 4 the open arrows of B point to the vertices v_i of P. The filled arrows demarcate the sets $e_i - v_i$; these sets are the "nonfirst" vertices of edges e_i of P. When P is nonempty we have $B[i] = i$ for $i = 1, 2$. Also when $i = k$ in (4) we interpret $B[2k+2]$ to be $\mathtt{TOP}(S) + 1$.

As in the strong components algorithm an array I stores stack indices as well as biconnected component numbers. More precisely for a given vertex v at any point in time,

$$(5) \qquad I[v] = \begin{cases} 0 & \text{if } v \text{ has never been in } P; \\ j & \text{if } v \text{ is currently in } P \text{ and } S[j] = v; \\ c & \text{if the last biconnected component containing } v \text{ has been output} \\ & \text{and numbered as } c. \end{cases}$$

As before there can be no confusion between an index j and a component number c in (5). A variable c is used to keep track of the component numbers.

The algorithm consists of a main routine `BICONN` and a recursive procedure `DFS`:

procedure $\mathtt{BICONN}(G)$

1. empty stacks S and B;
2. **for** $v \in V$ **do** $I[v] = 0$;
3. $c = n$;
4. **for** $v \in V$ **do if** $I[v] = 0$ and v is not isolated **then** $\mathtt{DFS}(v)$;

procedure $\mathtt{DFS}(v)$

1. $\mathtt{PUSH}(v, S)$; $I[v] = \mathtt{TOP}(S)$; **if** $I[v] > 1$ **then** $\mathtt{PUSH}(I[v], B)$; /* create a filled arrow on B */
2. **for** edges $\{v, w\} \in E$ **do**
3.   **if** $I[w] = 0$ **then** { $\mathtt{PUSH}(I[v], B)$; $\mathtt{DFS}(w)$ /* create an open arrow on B */ }
4.   **else** /* possible merge */ **while** $I[v] > 1$ and $I[w] < B[\mathtt{TOP}(B) - 1]$ **do** {$\mathtt{POP}(B)$; $\mathtt{POP}(B)$};
5. **if** $I[v] = 1$ **then** $I[\mathtt{POP}(S)] = c$
6. **else if** $I[v] = B[\mathtt{TOP}(B)]$ **then** {
7. $\mathtt{POP}(B)$; $\mathtt{POP}(B)$; increase c by 1;
8.   **while** $I[v] \le \mathtt{TOP}(S)$ **do** $I[\mathtt{POP}(S)] = c$ };

In many situations line B4 can be simplified: If G is known to be a connected graph B4 can be replaced by a single call $\mathtt{DFS}(v)$ (for any vertex v). If G has no isolated vertices, i.e., every vertex is on at least one edge, the second part of the **if** test of B4 can be dropped. Also moving the code for `DFS` for the case $I[v] = 1$ into B4 allows `DFS` itself to be simplified.

Theorem 3.1 *When* $\mathtt{BICONN}(G)$ *halts any edge* $\{v, w\} \in E$ *belongs to the biconnected component numbered* $\min\{I[v], I[w]\}$. *The time and space are both* $O(m + n)$.

Proof: The argument is similar to Theorem 2.1 and uses the conventions introduced in that proof. We prove the first assertion of the theorem by showing that BICONN is a valid implementation of the high-level algorithm.

We first specify how the high-level algorithm chooses the pair $v, \{v, w\}$ to grow P. Say a vertex $w \in V$ becomes *active* the first time it gets added to P. As before the *most active* vertex is the currently active vertex that was activated most recently. To choose $v, \{v, w\}$, let v be the most active vertex. Choose a previously unchosen edge $\{v, w\}$. If all edges incident to v have been chosen then deactivate v. If P is nonempty and this makes all vertices of $e_k - v_k$ inactive then output e_k as the next edge of the block hypergraph.

Note that it is possible to have P empty and a vertex v active. This can occur if v was the previous first vertex of P. In this case the above strategy starts a new path $P = (v, \{v, w\})$ by adding an edge incident to v. On the other hand it is possible to have P empty and no vertex v active. In this case when a new path $P = (v, \{v, w\})$ is started, by convention v becomes active before w. This convention ensures that the most active vertex is always unique.

This strategy correctly implements the high-level algorithm. To prove this we need only check that when P is nonempty $v \in e_k - v_k$, for v the most active vertex.

We will use another property of the implementation: When it chooses the pair $v, \{v, w\}$ to grow P, if $w \in V(P) - e_k$ then w is currently active. To show this note that $w \in V(P)$ implies w has been activated. Also $w \notin e_k$ implies $w, \{w, v\}$ has not been chosen to grow P (since after an edge is chosen, its ends belong to a common edge of P). This implies w is still active.

Now we prove that BICONN implements the above version of the high-level algorithm. We assume that H, P and the deleted biconnected components are as specified by (3)–(5). The argument is by induction on the number of statements executed in BICONN. We only mention the most important points about the various statements, leaving the remaining details of the induction to the reader.

When B4 is being executed, P is empty. During the execution of the loop of D2, v is the most active vertex. (Note that if $I[v] = 1$ then P is empty during the execution of D2. In this case we also have $\mathtt{TOP}(S) = 1$ and $\mathtt{TOP}(B) = 0$.)

In D3 suppose $I[w] > n$. Then the last biconnected component containing w has been deleted and D4 does nothing. Suppose $0 < I[w] \le n$. If $w \in e_k$ then $I[w] \ge I[v_k] = B[2k-1]$ (by (3)) so D4 does nothing. In the remaining case choose index i so $w \in e_i - v_{i+1}$, $1 \le i < k$. As noted for the high-level algorithm, w is an active vertex in $V(P)$. Our choice rule implies v_{i+1} is the most recently activated vertex of e_i that is still active. Thus $I[v_{i+1}] > I[w] \ge I[v_i]$. Equivalently by (3), $B[2i+1] > I[w] \ge B[2i-1]$. This shows D4 merges the same cycle as the high-level algorithm.

The test of D6 checks whether or not the last edge e_k of P consists of v and its successors on S, plus vertex v_k. Hence D8 labels vertices according to (5).

For the time and space bounds observe that every nonisolated vertex is pushed onto S exactly once. It is also pushed onto an even entry of B at most once. Hence it is easy to see that the algorithm spends $O(1)$ time on each vertex or edge. □

Comparing our code to the algorithm of [25], our stack S (which has at most n entries) corresponds to a stack of edges (which has at most m entries). Our size n array I corresponds to a similar array that holds depth-first discovery numbers. Our stack B (which has at most $2n$ entries) corresponds to a size n array that holds LOWPT values.

Acknowledgments

We thank San Skulrattanakulchai for helpful suggestions.

Appendix A Bridge Components

Consider an undirected graph G. An edge whose deletion increases the number of connected components is a *bridge*. Equivalently a bridge is an edge that does not belong to any cycle. A connected component of the graph formed by deleting all the bridges of G is a *bridge component* of G (also called a *2-edge-connected component*).

The *bridge component graph* is formed from G by contracting the vertices of each bridge component. Equivalently the bridge component graph is the acyclic graph, formed by contracting vertices of G, that has as many vertices as possible. It is the finest acyclic contraction of G. Note that the edges of the bridge component graph are precisely the (images of the) bridges of G.

The characterization of the bridge component graph is similar to the characterization of the strong component graph of Section 2. The high-level algorithm to find the bridge component graph is also similar. The main difference is that the working graph H, formed by contractions and deletions from G, is now a multigraph. That is parallel edges (perhaps created by contractions) are kept in H, since otherwise a nonbridge of G might become a bridge of H. This causes one change in the high-level algorithm for bridge components: When choosing an edge $\{v_k, w\}$ incident to the last vertex of P, do not choose the edge $\{v_k, v_{k-1}\}$ that is already in P. (However a parallel edge joining these vertices can be chosen.)

The low-level algorithm is similar to `STRONG`. It uses the same data structure, as specified by (1)–(2). When the algorithm halts an edge $\{v, w\} \in E$ is a bridge of G exactly when $I[v] \neq I[w]$.

For convenience the pseudocode below assumes the input graph G does not have parallel edges. `DFS` has two arguments: In $\texttt{DFS}(v, u)$, u is the vertex that calls $\texttt{DFS}(v, u)$. Aside from that only line `D4` differs from Section 2.

procedure $\texttt{BRIDGE}(G)$

1. empty stacks S and B;
2. **for** $v \in V$ **do** $I[v] = 0$;
3. $c = n$;
4. **for** $v \in V$ **do if** $I[v] = 0$ **then** $\texttt{DFS}(v, 0)$;

procedure $\texttt{DFS}(v, u)$

1. $\texttt{PUSH}(v, S)$; $I[v] = \texttt{TOP}(S)$; $\texttt{PUSH}(I[v], B)$; /* add v to the end of P */
2. **for** edges $\{v, w\} \in E$ **do**
3.   **if** $I[w] = 0$ **then** $\texttt{DFS}(w, v)$
4.   **else if** $w \neq u$ **then** /* contract if necessary */ **while** $I[w] < B[\texttt{TOP}(B)]$ **do** $\texttt{POP}(B)$;

if $I[v] = B[\texttt{TOP}(B)]$ **then** { /* number vertices of the next bridge component */
$\quad$ $\texttt{POP}(B)$; increase c by 1;
$\quad$ **while** $I[v] \leq \texttt{TOP}(S)$ **do** $I[\texttt{POP}(S)] = c$ };

As in Section 2 this algorithm runs in linear time.

Based on the similarity of our bridge and strong component algorithms, one can give an immediate proof of Robbins' Theorem [22]: A connected undirected graph has a strongly connected orientation if and only if it is bridgeless.

Appendix B Characterization of the Block Hypergraph

Lemma B.1 *The block hypergraph of a graph G is the finest acyclic merging of G.*

Proof: We first show the block hypergraph H is acyclic. A biconnected component of G is a connected subgraph of G. Hence a cycle in H gives a cycle in G that contains edges from at least two distinct biconnected components. This is impossible.

To show H is the finest acyclic merging let K be an acyclic merging of G. Any cycle of G is contained entirely in one edge of K. Thus any biconnected component is contained in one edge of K. $\square$

Appendix C Articulation Points

Consider an undirected graph G. A vertex whose deletion increases the number of connected components is an *articulation point*. We start with a high-level algorithm that finds the articulation points. The algorithm uses a working graph H that results from deleting some vertices and adding some edges to G. The degree of a vertex v in H is denoted $d(v)$. Initially H is G.

If H has no edges stop. Otherwise start a path P by choosing an edge $\{v, w\}$ and setting $P = (v, w)$. Continue by growing P as follows.

To grow the path $P = (v_1, \ldots, v_k)$ choose an edge $\{v_k, w\}$ with $w \notin P$ and do the following:

If the desired edge exists add w to P, making it the new last vertex. Continue growing P.

If the desired edge does not exist proceed as follows:

If $d(v_k) = 1$ and $d(v_{k-1}) > 1$ then mark v_{k-1} as an articulation point. (Note that $d(v_{k-1}) > 1$ always holds if $k > 2$.) Delete v_k from H and P. Continue growing P.

If $d(v_k) = d(v_{k-1}) = 1$ delete v_{k-1} and v_k from H and P. (Note that $k = 2$ in this case.) Try to start a new path P.

If $d(v_k) > 1$ (the remaining case) let v_i be the neighbor of v_k that has minimum index i. Delete v_k from both H and P. Add edge $\{v_{k-1}, v_i\}$ to H unless it already exists. Continue growing P.

To prove this algorithm correct we need only justify the last case, where $d(v_k) > 1$ and every neighbor of v_k belongs to P. Observe that the articulation points of a graph do not change if we subdivide an edge of a cycle or add a chord of a cycle. These two operations enable us to undo the changes that the algorithm makes to H. Hence the algorithm does not change the articulation points.

We discuss two natural low-level implementations. Both implementations number the vertices as they are added to P (i.e., the ith call $\mathtt{DFS}(v)$ numbers v as i).

The first implementation is equivalent to Tarjan's LOWPOINT method [25]: Each vertex v maintains a value $L[v]$ equal to the neighbor of v in H that is closest to the start of P (among all neighbors seen so far). If $\mathtt{DFS}(v)$ calls $\mathtt{DFS}(w)$ and the final value of $L[w]$ is v then v is an articulation point, unless it is the first vertex of P and all its neighbors have been added to P.

The second implementation uses a stack S instead of array L. S contains the vertices v_i of $P - v_k$ for which no edge $\{v_r, v_s\}$, $r < i < s$ has been scanned so far. Thus S contains the vertices of P that are possible articulation points. It is also convenient to maintain a quantity $C[v]$ equal to the number of biconnected components seen so far that contain an edge incident to v. Thus v is an articulation point if and only if $C[v] > 1$ when the algorithm halts.

procedure $\mathtt{ART}(G)$

empty stack S;
for $v \in V$ **do** $C[v] = I[v] = 0$;
$i = 0$;
for $v \in V$ **do if** $I[v] = 0$ **then** $\mathtt{DFS}(v, 0)$;

procedure $\mathtt{DFS}(v, u)$

if $u \neq 0$ **then** $\mathtt{PUSH}(u, S)$; increase i by 1; $I[v] = i$; /* add v to the end of P */
for edges $\{v, w\} \in E$ **do**
    if $I[w] = 0$ **then** $\mathtt{DFS}(w, v)$
    else while $I[w] < I[\mathtt{TOP}(S)]$ **do** $\mathtt{POP}(S)$;
if $u \neq 0$ **then** {
  increase $C[v]$ by 1;
  if $u = S[\mathtt{TOP}(S)]$ **then** { increase $C[u]$ by 1; $\mathtt{POP}(S)$ } };

Appendix D Space Savings

Bob Tarjan has suggested a strong component algorithm that replaces S and B by two other stacks, such that a given vertex is in at most one stack at a time. This allows the stacks to share the same n locations, with one stack growing up and the other down. (Eve and Kurki-Suonio [8] present a LOWPOINT-based strong component algorithm that uses the same amount of space.) We give a brief description of Tarjan's algorithm in our terminology.

As in the previous section the vertices of G are numbered in the order they are added to P. Suppose $P = (v_1, \ldots, v_k)$. The stack B contains k elements f_i, $i = 1, \ldots, k$, where f_i is the first vertex of v_i that was added to P. The stack R contains the remaining vertices of G in P. Vertices are in R in correct order, i.e., vertices of v_i precede vertices of v_j for $j > i$. It is convenient to allow a vertex of $v_i - f_i$ not to be in R: It gets pushed onto R in $\mathtt{DFS}(v)$ if the test of D5 of $\mathtt{STRONG}$ fails.

This causes no problem since the only purpose of R is to allow vertices to be numbered when their strong component is output.

After seeing this algorithm we noticed another way to reduce the extra space by n which we now describe. In `STRONG` we do not allocate space for the B stack. Instead a variable tb maintains the top entry in B (i.e., the vertex f_k above). The rest of the B stack is stored in I, i.e., if j is an entry in the B stack then $I[S[j]]$ is the preceding entry. The algorithm is as follows.

procedure `STRONG`(G)
empty stack S; $tb = -1$;
for $v \in V$ **do** $I[v] = 0$;
$c = n$;
for $v \in V$ **do if** $I[v] = 0$ **then** `DFS`(v);

procedure `DFS`(v)
`PUSH`(v, S); $I[v] = tb$; $tb =$ `TOP`(S); /* add v to the end of P */
for edges $(v, w) \in E$ **do**
  if $I[w] = 0$ **then** `DFS`(w)
  else /* contract if necessary */ **while** $I[w] < tb$ and $w \neq S[tb]$ **do** $tb = I[S[tb]]$;
if $v = S[tb]$ **then** { /* number vertices of the next strong component */
  $old\_tb = tb$; $tb = I[v]$; increase c by 1;
  while $old\_tb \leq$ `TOP`(S) **do** $I[$`POP`$(S)] = c$ };

A similar strategy can be used to reduce the size of the B stack in `BICONN` from $2n$ to n words.

References

[1] A.V. Aho, J.E. Hopcroft and J.D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading MA, 1974.

[2] A.V. Aho, J.E. Hopcroft and J.D. Ullman, *Data Structures and Algorithms*, Addison-Wesley, Reading MA, 1983.

[3] C. Beeri, R. Fagin, D. Maier, A. Mendelzon, J. Ullman, M. Yannakakis, "Properties of acyclic database schemes," *Proc. 13th Annual ACM Symp. on Theory of Comp.,* 1981, pp. 355–362.

[4] C. Berge, *Hypergraphs: Combinatorics of Finite Sets*, North-Holland, NY, 1989.

[5] G. Brassard and P. Bratley, *Algorithmics: Theory & Practice*, Prentice-Hall, Englewood Cliffs NJ, 1988.

[6] G. Brassard and P. Bratley, *Fundamentals of Algorithmics*, Prentice-Hall, Englewood Cliffs NJ, 1996.

[7] T.H. Cormen, C.E. Leiserson and R.L. Rivest, *Introduction to Algorithms*, McGraw-Hill, NY, 1990.

[8] J. Eve and R. Kurki-Suonio, "On computing the transitive closure of a relation," *Acta Informatica 8*, 1977, pp. 303–314.

[9] S. Even, *Graph Algorithms*, Computer Science Press, Potomac, MD, 1979.

[10] H.N. Gabow, "Path-based depth-first search for strong and biconnected components," *Information Processing Letters*, to appear.

[11] H.N. Gabow and R.E. Tarjan, "A linear-time algorithm for a special case of disjoint set union," *J. Comp. and System Sci., 30,* 2, 1985, pp. 209–221.

[12] J.E. Hopcroft and R.E. Tarjan, "Dividing a graph into triconnected components," *SIAM J. Comput., 2,* 1973, pp. 135–158.

[13] J.E. Hopcroft and R.E. Tarjan, "Efficient planarity testing," *J. ACM, 21,* 4, 1974, pp. 549–568.

[14] F. Harary, *Graph Theory*, Addison-Wesley, Reading MA, 1969.

[15] E. Horowitz, S. Sahni, S. Rajasekaran, *Computer Algorithms*, Computer-Science Press, NY, 1998.

[16] D.E. Knuth, *The Stanford Graphbase: A Platform for Combinatorial Computing*, Addison-Wesley, Reading MA, 1993.

[17] L. Lovász, *Combinatorial Problems and Exercises*, 2nd Edition, North-Holland, NY, 1993.

[18] U. Manber, *Introduction to Algorithms: A Creative Approach*, Addison-Wesley, Reading MA, 1989.

[19] K. Melhorn, *Data Structures and Algorithms 2: Graph Algorithms and NP-Completeness*, Springer-Verlag, NY, 1984.

[20] I. Munro, Efficient determination of the strongly connected components and transitive closure of a directed graph, Department of Computer Science, University of Toronto, 1971.

[21] P.W. Purdom, A transitive closure algorithm, Tech. Rept. 33, Computer Sciences Department, University of Wisconsin, Madison, 1968.

[22] H.E. Robbins, "A theorem on graphs with an application to a problem of traffic control," *American Math. Monthly, 46*, 1939, pp. 281–283.

[23] R. Sedgewick, *Algorithms in C*, Addison-Wesley, Reading MA, 1990.

[24] M. Sharir, "A strong-connectivity algorithm and its application in data flow analysis," *Computers and Mathematics with Applications 7*, 1, 1981, pp. 67–72.

[25] R.E. Tarjan, "Depth-first search and linear graph algorithms," *SIAM J. Comput., 1,* 2, 1972, pp. 146–160.

[26] R.E. Tarjan, "Efficiency of a good but not linear set union algorithm," *J. ACM, 22,* 2 1975, pp. 215–225.

[27] R.E. Tarjan, personal communication, handwritten notes.

[28] M.A. Weiss, *Data Structures and Algorithm Analysis in C++*, Addison-Wesley, Reading MA, 1999.