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

Harold N. Gabow

CU-CS-890-99

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

Harold N. Gabow *

October 15, 1999

Abstract

The known 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 [23]. This note presents depth-first search algorithms for strong and biconnected components that are not based on the depth-first search tree but rather the depth-first search path. This simple view should make the algorithms easier to code and maintain. The algorithms appear to be just as efficient as traditional depth-first search algorithms.

Most depth-first search algorithms (e.g., [23, 11, 12]) 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 components algorithms [15, pp. 94, 514]. Tarjan's LOWPOINT method for strong components is presented in texts [1, 8, 15, 17, 18, 21]. The strong component algorithm of Kosaraju and Sharir [22] is often regarded as easier to understand but involves two passes over the graph; it is presented in texts [2, 5, 7, 25]. Tarjan's LOWPOINT biconnected component algorithm is presented in texts [1, 2, 5, 6, 8, 14, 15, 17, 18, 21, 25]. A two-pass biconnected component algorithm of Micali that avoids LOWPOINT values is sketched in [8, pp.67-68].

This paper presents versions of the strong and biconnected components algorithms that are based on the depth-first search path. This natural approach was first proposed by Purdom [20] and Munro [19] for strong connectivity. It is commonly regarded as requiring an extra data structure for set merging in order to be asymptotically efficient, and hence unlikely to be efficient in practice [23]. We present linear-time implementations of the method for strong and biconnected components, using only stacks and arrays. A line-by-line pseudocode comparison with the tree-based algorithms

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

of [23] shows the two approaches are similar in terms of lower level resource usage; performance differences are likely to be small or platform-dependent. Our algorithm shows that the simpler path-based view of depth-first search suffices for these properties. Our motivation in developing these algorithms was the need to compute strong components in an implicitly defined graph where maintaining exact LOWPOINT values would be inefficient [9].

It is possible to derive other path-based depth-first search algorithms. For instance the ear decomposition [16] of a 2-edge-connected, biconnected or strongly connected digraph can be found in linear time. This can be used to find an st-numbering [8] of a biconnected graph. The algorithm of Section 2 gives a topological numbering of a dag. For more involved graph properties like triconnectivity and planarity it is likely that a tree-based approach with LOWPOINT numbers is necessary.

Section 2 presents our strong components algorithm and Section 3 presents the biconnected components algorithm. Appendix A proves a simple property of biconnected components. 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 use the following operations to manipulate a stack S: `PUSH`(x, S) adds x to S at the (new) top of S. `POP`(S) removes the value at the top of the stack and returns that value. `TOP`(S) is the index of the value at the top of the stack. Hence $S[$`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 finest acyclic contraction of G, i.e., it is the acyclic digraph formed by contracting vertices of G that has as many vertices as possible.

The latter characterization suggests the following method to find the strong component graph of $G = (V, E)$. See Fig. 1(a)–(b). The method maintains a graph that is a contraction of G and a path P in that graph.

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

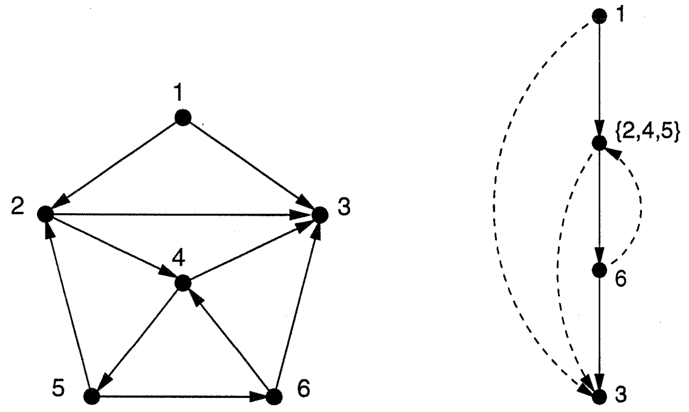
To grow the path $P = (v_1, \ldots, v_k)$ choose an edge (v_k, w) directed from the last vertex of P and do the following:

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

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

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

It is clear from our description of strong components that this algorithm stops having correctly output all strong components of the original graph.



(a) (b)

i	1	2	3	4	5	6
$S[i]$	1	2	4	5	6	3
$B[i]$	1	2	5	6		
$I[i]$	1	2	6	3	4	5

(c)

Figure 1: (a) Digraph. (b) P (solid edges) and contracted graph (all edges) right before strong component $\{3\}$ is deleted. (c) Data structure corresponding to (b).

This high-level algorithm was originally proposed by Purdom [20] and Munro [19]. An efficient implementation requires a data structure to keep track of the strong components as they are formed by contraction operations. The data structures of [20] and [19] use time $O(n^2)$ and $O(n \log n)$ for this task respectively. Any data structure for disjoint set merging [7] can be used to keep track of the strong components. Tarjan showed the set merging can be done in time $O(m\alpha(m,n))$ [24]. In fact the incremental tree algorithm of [10] 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. Furthermore 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. 1(c).

We assume the vertices 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 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 $i = 1, \ldots, k$,

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

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,

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

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 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 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 vertices */ while $B$[TOP($B$)] $> I[w]$ do POP($B$);
5.  if $B$[TOP($B$)] $= I[v]$ then { /* number vertices of the next strong component */
6.      POP($B$); increase $c$ by 1;
7.      while TOP($S$) $\geq I[v]$ do { $w$ = POP($S$); $I[w] = c$ } };
```

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 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. For the first assertion of the theorem we prove the following properties hold throughout the execution of STRONG:

(*i*) At all times the current graph, the path P and the deleted strong components are as specified by (1)–(2).

(*ii*) Whenever D2 or D5 is executed v belongs to the last vertex of P.

We prove (*i*)–(*ii*) by induction on the number of statements executed. After that we show the first assertion of the theorem follows easily. Now we give the inductive argument.

Suppose DFS(v) is called from S4. We show D1 makes $P = (v)$ according to (1)–(2). Inspection of D1 shows it suffices to have $P = \emptyset$ immediately before DFS(v). This is obvious for the first call that S4 makes. For a subsequent call let DFS(u) be the preceding call made from S4. (*ii*) shows that when D5 is executed in DFS(u), u belongs to the last vertex of P. Since $I[u] = 1$, D7 deletes the strong component containing u and makes $P = \emptyset$.

Suppose DFS(v) is called from D3 in DFS(u). We show D1 adds v to the end of P and (*i*) holds. The test of D3 shows v is not in P when DFS(v) is called (by (*i*)). Since (*ii*) shows u belongs to the last vertex of P, (*i*) holds after D1.

Next consider an edge (v, w) chosen in D2. We have already checked the case $I[w] = 0$.

Suppose $0 < I[w] \leq n$. Then w is in a vertex of P. D4 contracts the correct cycle, preserving (*i*). (If (v, w) has already been contracted then D4 does nothing.) Furthermore (*ii*) holds for the next execution of D2.

Suppose $I[w] > n$. Then the component containing w has been deleted. The test of D4 fails, so STRONG does nothing for this edge.

Next consider the test of D5. It checks whether or not the last vertex of P consists of v and its successors on S, by (i). Suppose the test is true. All edges incident to the last vertex of P have been processed. Hence the last vertex of P is (the contraction of) a strong component. D7 correctly numbers and deletes the strong component, preserving (i).

Finally suppose DFS(v) was called in line D3 of DFS(u). We must show (ii) holds for u when DFS(v) returns. By induction (ii) shows u was in the last vertex of P when DFS(v) was called. The vertex containing u immediately precedes the vertex containing v in P, until either the latter is deleted in DFS(v), or the two vertices are contracted. In both cases (ii) holds for u when DFS(v) returns.

This completes the induction. Now observe that DFS(v) is called for every vertex v (by the loop of S4). Hence every v is added to P at some point. We have already seen that each call to DFS(v) in S4 ends with $P = \emptyset$. Hence every vertex gets numbered with its correct strong component, by (i).

For the time bound it is easy to see the algorithm spends $O(1)$ time on each vertex or edge, assuming the graph is stored as a collection of adjacency lists. Note that every vertex is pushed onto and popped from each stack S, B exactly once. $\square$

Comparing our code to the algorithm of [23], both methods use stack S; our size n array I corresponds to a similar array that holds depth-first discovery numbers; and our stack B corresponds to a size n array that holds LOWLINK values.

3 Biconnected Components

We present our algorithm for biconnected components using the language of hypergraphs. While this is not logically necessary it brings out the similarity to the strong components algorithm.

We start by reviewing basic definitions about hypergraphs [4, 16]. 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 $v_1, e_1, \ldots, v_k, e_k$, $k \geq 1$, of distinct vertices v_i and distinct edges e_i, $1 \leq i \leq k$, with $v_1 \in e_1$ and $v_i \in e_{i-1} \cap e_i$ for every $1 < i \leq k$. By convention a sequence of one vertex v_1 is also a path. The set of vertices in P is denoted $V(P) = \bigcup_{i=1}^{k} e_i$. A *cycle* is a sequence satisfying the above definition of 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].)

We use this operation on hypergraphs: To *merge* edges e_i, $i = 1, \ldots, k$, add a new edge $\bigcup_{i=1}^{k} e_i$ and delete every edge properly 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 edges are in the same *biconnected component* of G if and only if some simple cycle contains both of them. It is easy to see the biconnected components are well-defined since being on the same simple cycle is an equivalence relation over the edges. The "block-cutpoint tree" of a graph represents the biconnected components and cutpoints [13]. We define 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 A.

The characterization suggests the following method to find the block hypergraph of $G = (V, E)$. See Fig. 2(a)–(b). The method maintains a hypergraph that is a merging of G and a path P in that hypergraph.

If the hypergraph has no edges stop. Otherwise start a 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.

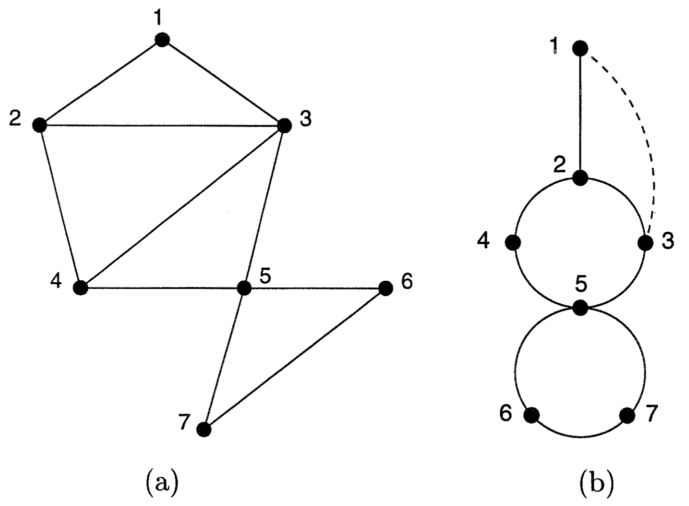
To grow the path $P = (v_1, e_1, \ldots, v_k, e_k)$ choose an edge $\{v, w\}$ with $v \in e_k - v_k$ and do the following:

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 $w, e_i, v_{i+1}, e_{i+1}, \ldots, v_k, e_k, \{v, w\}$ to a new edge $e = \bigcup_{j=i}^{k} e_j$. P is a path ending with e in the new hypergraph (i.e., $(v_i, e_i, \ldots, v_k, e_k)$ is replaced by (v_i, e)). 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 the hypergraph 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 method is based on two simple observations: When $v, \{v, w\}$ is added to P the result is a path by the condition $v \in e_k - v_k$. Edges that are merged are guaranteed to form a cycle by the condition $w \in e_i - v_{i+1}$. Now a simple inductive argument proves the method correctly outputs the edges of the block hypergraph.



i	1	2	3	4	5	6	7
$S[i]$	1	2	3	5	4	7	6
$B[i]$	1	2	4	6			
$I[i]$	1	2	3	5	4	7	6

(c)

Figure 2: (a) Undirected graph. (b) P (solid edges) and hypergraph (all edges) right before edge $\{5, 6, 8\}$ is deleted. (c) Data structure corresponding to (b).

As in Section 2 we now give a simple list-based implementation that achieves linear time. The data structure is illustrated in Fig. 2(c). As before assume the vertices are numbered by consecutive integers from 1 to n. The algorithm numbers the biconnected components of G by consecutive integers starting at $n+1$. We represent the biconnected components by assigning a number $I[v]$ to each vertex v, so 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 sequence of vertices in P and stack B represents the boundaries between edges, two vertices for each boundary. More specifically S and B correspond to $P = (v_1, e_1, \ldots, v_k, e_k)$, $k \geq 1$, where $\mathtt{TOP}(B) = 2k$ and for $i = 1, \ldots, k$,

$$v_i = S[B[2i-1]]; \tag{3}$$

$$e_i = v_i \cup \{S[j] : B[2i] \leq j < B[2i+1]\}. \tag{4}$$

When $k \geq 1$ we have $B[i] = i$ for $i = 1, 2$. Also when $i = k$ in (4) we interpret $B[2k+1]$ to be $\mathtt{TOP}(S) + 1$. Finally at certain points P is a path (v), in which case $S[1] = v$, $\mathtt{TOP}(S) = 1$ and $\mathtt{TOP}(B) = 0$.

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,

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

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 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 $I[v] = 0$ then { DFS($v$); POP($S$); $I(v) = c$ };

procedure DFS($v$)
1.  PUSH($v, S$); $I[v]$ = TOP($S$); if $I[v] > 1$ then PUSH($I[v], B$); /*$v$ is the second boundary vertex*/
2.  for edges $\{v,w\} \in E$ do
3.      if $I[w] = 0$ then { PUSH($I[v], B$); DFS($w$) /* $v$ is the first boundary vertex */ }
4.      else /* merge */ while $I[w] < B$[TOP($B$)] and $I[w] \neq B$[TOP($B$)$-1$] do {POP($B$); POP($B$)}
5.  if $I[v] > 1$ and $I[v] = B$[TOP($B$)] then {
6.      POP($B$); POP($B$); increase $c$ by 1;
7.      while TOP($S$) $\geq I[v]$ do { $u$ = POP($S$); $I[u] = c$ } }
```

Theorem 3.1 *When* 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: We refer to lines of pseudocode as in Theorem 2.1. For the first assertion of the theorem we prove the following properties hold throughout the execution of `BICONN`:

(i) At all times the current hypergraph, the path P and the deleted biconnected components are as specified by (3)–(5).

(ii) When `D2` or `D5` is executed, if $I[v] > 1$ then v belongs to the last edge e_k of P and v is not the preceding vertex v_k. If $I[v] = 1$ then $P = (v)$.

We prove (i)–(ii) by induction on the number of statements executed. After that we show the first assertion of the theorem follows. Now we give the inductive argument.

Suppose `DFS`(v) is called from `B4`. We show `D1` makes $P = (v)$ according to (3)–(4). Inspection of `D1` shows it suffices to have `TOP`(S) = `TOP`$(B) = 0$ immediately before `DFS`(v). This is obvious for the first call that `B4` makes. For a subsequent call let `DFS`(u) be the preceding call made from `B4`. (ii) shows that when `D5` is executed in `DFS`(u), $P = (u)$. Since `B4` pops S the desired equations hold.

Suppose `DFS`(v) is called from `D3` in `DFS`(u). We show `D3` and `D1` add $v, \{v, w\}$ to the end of P and (i) holds. The test of `D3` shows $v \notin V(P)$ when `DFS`(v) is called (by (i)). Now the inductive assumption of (ii) for u implies (i) holds after `D1`.

Next consider an edge $\{v, w\}$ chosen in `D2`. We have already checked the case $I[w] = 0$.

Suppose $0 < I[w] \le n$. Then $w \in V(P)$. `D4` merges the edges of the correct cycle, preserving (i). (If $\{v, w\}$ has already been deleted by a merge then `D4` does nothing.) Furthermore (ii) holds for the next execution of `D2`.

Suppose $I[w] > n$. Then the last biconnected component containing w has been deleted. The test of `D4` fails, so `BICONN` does nothing for this edge.

Next consider the test of `D5`. It checks whether or not the last edge e_k of P consists of v and its successors on S, plus the vertex $v_k = S[B[\texttt{TOP}(B) - 1]]$, by (i). Suppose the test is true. All graph edges incident to vertices of $e_k - v_k$ have been processed. Hence the last edge of P corresponds to a biconnected component. `D7` correctly numbers and deletes the component, preserving (i).

Finally suppose `DFS`(v) was called in line `D3` of `DFS`(u). We must show (ii) holds for u when `DFS`(v) returns. By induction (ii) shows u was in the last edge of P when `DFS`(v) was called. After `D1` of `DFS`(v), u belongs to two consecutive edges of P. This continues to hold until either the second edge is deleted in `DFS`(v), or the two edges are merged together. In both cases (ii) holds for u when `DFS`(v) returns.

This completes the induction. Now observe that `DFS`(v) is called for every vertex v (by the loop of `B4`). Hence every v is added to P at some point. We have already seen that each call to `DFS`(v) in `B4` ends by making $P = \emptyset$. Hence every vertex gets numbered with its correct biconnected component, by (i).

For the time bound it is easy to see that `BICONN` and `DFS` spend $O(1)$ time per vertex or edge, assuming the graph is stored as a collection of adjacency lists. Every vertex is pushed onto S exactly once. It is also pushed onto an even entry of B at most once. $\square$

Comparing our code to the algorithm of [23], our stack S corresponds to a stack of edges; our size n array I corresponds to a similar array that holds depth-first discovery numbers; and our stack B corresponds to a size n array that holds LOWPT values.

A Appendix

Lemma 1.1 *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, so H is acyclic.

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. □

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

[9] H.N. Gabow, "Applications of a poset representation to edge connectivity and graph rigidity," *Proc. 32nd Annual Symp. on Found. of Comp. Sci.,* 1991, 812-821.

[10] 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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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