Enumerating the Cycles of a Digraph: A New Preprocessing Strategy

G. LOIZOU

and

P. THANISCH

Department of Computer Science, Birkbeck College,
University of London, Malet Street, London WC1E 7HX, England

Communicated by A. M. Andrew

ABSTRACT

Szwarcfiter and Lauer have published one of the most efficient known algorithms for enumerating the cycles of a digraph. An improved version of this algorithm together with a new strategy for preprocessing the digraph, in order to simplify the cycle enumeration process, is presented.

1. INTRODUCTION

In a review article which appeared in 1976, Mateti and Deo [6] listed more than 20 papers containing algorithms for enumerating the cycles of a digraph. For example, Syslo's algorithm [12, 13] performs very well when the input digraph is *dense*, and furthermore it manages to combine elegance with nonrecursiveness. It does, however, have certain disadvantages when dealing with sparse digraphs. In common with many other cycle-finding algorithms, this algorithm investigates the digraph by extending and retracting a path kept on a stack. One of the algorithm's drawbacks is that whenever a vertex v is added to this stack, a check is made to see if there is an arc leading into v from each of the n vertices of the digraph. The sparser the digraph, therefore, the lower will be the average indegree of the vertices, and thus the less frequently will this checking actually find an arc. Moreover, its *worst-case* time complexity is exponential.

On the other hand, two of the advantages that Read and Tarjan [9] claim for their algorithm diminish when the digraph to be searched is dense. Firstly, their algorithm performs a preliminary depth-first search on the digraph to find the set of *cycle start vertices*, that is, the set of vertices that have incoming arcs (called *cycle arcs*) from their descendants on the depth-first search tree. They show that each cycle can be thought of as ending in a cycle arc. Thus, their recursive, backtracking procedure for extending and retracting the path under investigation is only called from the main program once for each cycle start vertex. The trouble is that in a dense digraph nearly all vertices will be cycle start vertices. Another advantage that Read and Tarjan claim for their algorithm is that when a vertex is to be added to the current path under investigation, a depth-first search is performed to discover (for future backtracking purposes) whether some alternative extension to the path is possible at this stage; only when such a choice exists will the investigation procedure be called recursively. Again, the trouble is that in a dense digraph there will almost always be such a choice.

From the foregoing, it might be concluded that in any application requiring a cycle-finding algorithm, a collection of such algorithms should be available. Then when a digraph is read in, Syslo's algorithm, for example, could be used if there is a high ratio of arcs to vertices, Read and Tarjan's algorithm could be used if there is a low ratio of arcs to vertices, and a third algorithm could be used for intermediate cases.

A possible candidate for this third algorithm is the one published by Szwarcfiter and Lauer [14, 15]. In Section 3 we propose a modification of this algorithm. Apart from reducing both the algorithm's storage requirements and the number of instructions that will be executed for a given input digraph, the proposed modification also makes the action of the algorithm easier to understand.

Several published algorithms (see, for example, Weinblatt [19] and Szwarcfiter and Lauer [15]) have taken advantage of the fact that all the arcs of a cycle must lie in the same strongly connected component (hereinafter abbreviated to scc for convenience) of the digraph. Tsukiyama et al. [17] have gone one stage further and have taken advantage of the fact that all arcs of a cycle must lie in the same strongly connected block. In Section 4 we demonstrate the advantage of breaking the digraph down into its constituent strongly connected blocks and passing each of these in turn to the modified Szwarcfiter-Lauer cycle enumeration algorithm.

It should be noted at this point that the aforesaid preprocessing can also enhance the efficiency of other algorithms. For example, the algebraic algorithms by Srimani [10] and Srimani and Sengupta [11] would benefit in efficiency, with respect to both space and time, were they to search a digraph's strongly connected blocks for cycles, taking as input the blocks one at a time, rather than search the whole digraph unpreprocessed. Srimani and Sengupta's algorithm selects a *start vertex* and then performs a breadth-first search on the digraph, simultaneously storing each path which emanates from the start vertex. Srimani and Sengupta refer to this process as *expanding a reachability equation*, and claim that "*the number of times the operation of expanding a reachability equation is to be executed is equal to the length of the maximum length directed circuit present in the graph*" [11, p. 1413]. This statement is false, since in a digraph with more than one strongly connected block, the longest cycle cannot be longer than the number of vertices in the largest strongly connected block, yet the number of times that the reachability equation will have to be expanded, for a *given* start vertex, will depend on the length of the longest *directed* path emanating from the start vertex. This path could even be Hamiltonian.

It has been noted in several recent papers (see, for example, Pippenger [8]) that there is a time-space tradeoff between different algorithmic solutions to a given problem. In Section 5 we propose a cycle-finding algorithm with a space bound that is exponential in the size of the digraph. While this algorithm has the same worst-case time complexity as the Szwarcfiter-Lauer algorithm (or its modified form), the performance of the proposed algorithm *can* be far superior to that of the said algorithm and all its predecessors. It should be noted here that the algorithm in [11] also has an exponential space bound.

In the next section we give, for the sake of the reader not familiar with the varied terminology of digraph theory, the basic definitions required in the ensuing development.

2. NOTATION AND DEFINITIONS

In building up the system of definitions, the term *vertex* is taken to be primitive. An *arc*, (v, w), is an ordered pair of *distinct* vertices. Multiple arcs as well as self-loops are not considered in the present paper. A *digraph* D is composed of a finite, nonempty set V of vertices and a set E of arcs. Every vertex in the ordered pairs that make up E is an element of the set V. A digraph with just one vertex in its vertex set is said to be *trivial*. Any other digraph is said to be *nontrivial*.

Let $D = (V, E)$ be a digraph. The notation $D' = D - S$ refers to the formation of a new digraph, $D' = (V', E')$, by removing from V a set S $(S \subset V)$ of vertices, and removing from E all the arcs incident with vertices in S. D' is called a *subdigraph* of D.

The cardinality of the vertex set V, which as usual we denote by $|V|$, is taken to be n, and the vertices in V are represented by the integers 1 to n. The cardinality of the arc set E is assumed to be e, i.e., $|E| = e$.

An *undirected graph* (usually just called a *graph*) is a special case of a digraph where the existence of an arc (v, w) implies the existence of an arc (w, v).

The *indegree* of a vertex v is the number of arcs in E which are *incident into* v.

Correspondingly the *outdegree* of v is the number of arcs in E which are *incident out of* v.

A *path* p in a digraph $D=(V,E)$ is a sequence of distinct vertices $v_1, v_2, \ldots, v_{k-1}, v_k$ such that:

(1) $v_i \in V\ (1 \leqslant i \leqslant k)$,
(2) $v_i = v_j$ if and only if $i=j\ (1 \leqslant i, j \leqslant k)$,
(3) $(v_i, v_{i+1}) \in E\ (1 \leqslant i \leqslant k-1)$.

A *cycle* in a digraph $D=(V,E)$ is a sequence of vertices $v_1, v_2, \ldots, v_k$ such that:

(1) $v_i \in V\ (1 \leqslant i \leqslant k)$,
(2) $v_1, v_2, \ldots, v_{k-2}, v_{k-1}$ is a path,
(3) $(v_{k-1}, v_k) \in E$,
(4) $v_1 = v_k$,
(5) $k \geqslant 3$.

In the sequel reference is made to *concatenation* of paths. The result of the concatenation operation can either be a path or a cycle. Consider two paths, $p_1 = u_1, u_2, \ldots, u_{k-1}, u_k$ and $p_2 = v_1, v_2, \ldots, v_{m-1}, v_m$. Path p_2 can only be concatenated to path p_1 if the following conditions are satisfied:

(1) $u_k = v_1$,
(2) the sequence $u_1, u_2, \ldots, u_{k-1}, v_1, v_2, \ldots, v_{m-1}, v_m$ is either a path $(u_1 \neq v_m)$ or a cycle $(u_1 = v_m)$.

In the present paper the symbols "$\oplus$" and "$\underline{\oplus}$" are used to denote concatenation operations. Their meanings are as follows:

(1) $q := p_1 \oplus p_2$, where p_1 and p_2 are paths, means that q becomes p_2 concatenated to p_1. The operation is only defined whenever the result q becomes a valid path or cycle.

(2) $Q := P_1 \underline{\oplus} P_2$, where P_1 and P_2 are sets of paths, means that Q becomes the set of paths or cycles formed by concatenating each member of the set P_2 to each member of the set P_1. The operation is only defined whenever each individual concatenation operation produces a valid path or cycle. Note that $|Q| \leqslant |P_1|\,|P_2|$.

A *semipath* in a digraph $D=(V,E)$ is a sequence of vertices $v_1, v_2, \ldots, v_{k-1}, v_k$ such that:

(1) $v_i \in V\ (1 \leqslant i \leqslant k)$,
(2) $v_i = v_j$ if and only if $i=j\ (1 \leqslant i, j \leqslant k)$,
(3) $(v_i, v_{i+1}) \in E$ or $(v_{i+1}, v_i) \in E\ (1 \leqslant i \leqslant k-1)$.

A *connected* digraph is a digraph in which every pair of distinct vertices is joined by a semipath. The *maximal* connected subdigraphs of a nonconnected digraph are known as its (connected) *components*.

A nontrivial digraph D is *strongly connected* if and only if for each pair of distinct vertices $\{v, w\}$ there is a path from v to w (v-w path) and a path from w to v (w-v path). The trivial digraph is defined to be strongly connected. A maximal strongly connected subdigraph D' of a digraph D is called an scc of D. The strong connectedness criterion partitions the vertex set V into equivalence classes. (The criterion does not partition the arc set E, as some arcs may go from one scc to another.)

Suppose that the strong connectedness criterion partitions V into r classes. Call these classes V_i $(1 \leqslant i \leqslant r)$. Define r sets E_i $(1 \leqslant i \leqslant r)$ such that, if $(v, w) \in E$ and for some i $(1 \leqslant i \leqslant r)$ $v, w \in V_i$, then $(v, w) \in E_i$. The sccs of the digraph D can now be defined as the r subdigraphs $D_i = (V_i, E_i)$ $(1 \leqslant i \leqslant r)$.

A *cutpoint* is a vertex v in a digraph D whose removal from D leaves a digraph $D' = D - \{v\}$ with a larger number of connected components than D. A nontrivial digraph is said to be *biconnected* if it has no cutpoints. A *block* is a maximal biconnected component of a digraph. Note that a block need not be strongly connected and that a strongly connected digraph need not be a block.

A *separation pair* is a pair of vertices $\{v, w\}$ whose removal from a strongly connected block D leaves a digraph $D' = D - \{v, w\}$ which comprises more than one connected component. The existence of a separation pair can be used to define an equivalence relation on the set E. In this equivalence relation the arcs of E are divided into equivalence classes $E_1, E_2, \ldots, E_r$ such that two arcs which lie on a common path not containing vertex v or vertex w (i.e., the vertices of the separation pair)—except as an endpoint—are in the same equivalence class. The E_i $(1 \leqslant i \leqslant r)$ are called the *separation classes* of D with respect to $\{v, w\}$.

A digraph is *triconnected* if it is a strongly connected block and it does not contain a separation pair. A *triconnected component* D_t of a digraph D is a maximal strongly connected block of D that does not contain a separation pair.

3. THE MODIFIED SZWARCFITER-LAUER ALGORITHM

The Szwarcfiter-Lauer algorithm [15] works as follows:

1. *Input and preprocessing*: A digraph D is read in. The nontrivial sccs of D are found, and the cycle-finding stage of the algorithm is applied to each scc in turn.

2. *Cycle finding*: A vertex v_1 with the largest indegree is selected as the start vertex, marked, and put on a path stack. By the time v_1 is unstacked, all the cycles in the current scc will have been enumerated and the next scc (if there is one) can be searched. The algorithm searches the scc by extending and retracting the path stack, using a recursive backtracking procedure.

While a vertex v is on the stack, every vertex in its adjacency list $A(v)$ [1] will be explored once. The search procedure keeps a record of those vertices in $A(v)$ which have yet to be explored during v's current sojourn on the stack. When v becomes the vertex on top of the stack (this happens either when it has just been stacked or when the vertex that was above it on the stack has just been popped), an attempt is made to select the next unexplored vertex in $A(v)$. If this adjacency list has been exhausted, then the path under exploration is retracted and v is removed from the stack. Otherwise the next unexplored vertex w in the adjacency list $A(v)$ is explored.

The algorithm's course of action at this stage depends on the status of w:

(1) If w is unmarked, then the path under examination is extended to w; w is put on the stack and marked. Eventually, the path under examination will be retracted and v will again become the top vertex on the stack. At this stage, a check is made to see if any new cycles were enumerated while w was on the stack. If not, the vertex w is removed from $A(v)$.

(2) If w is marked, then a check is made to discover if w is on the stack, and there is at least one vertex above w which has not yet been removed from the stack. If this proves to be the case, a cycle, comprising the sequence of vertices on the stack from w up to v, has been found, and this sequence of vertices is output. Otherwise, the exploration of arc (v, w) has been fruitless, and vertex w is removed, for the time being, from $A(v)$.

Once the adjacency list of v, i.e. $A(v)$, has been exhausted, a check is made to see if any cycles were enumerated during v's sojourn on the stack. If not, v is left marked after it is removed from the stack. Otherwise, v is unmarked and any deleted vertices which lie on a path of deleted vertices leading to v are restored to their adjacency lists.

In the Szwarcfiter-Lauer algorithm the digraph is represented by the set of its adjacency lists, and the algorithm uses the following marking system:

B lists. If the exploration of an arc (v, w) does not lead to a new cycle being found, then w is removed from the adjacency list $A(v)$ and vertex v is inserted in list $B(w)$.

Position vector. If a vertex v is the jth vertex from the bottom of the stack, then position$(v) = j$. When v is deleted from the stack, position(v) is assigned the value $n+1$, where n is the number of vertices of the input digraph.

Reach vector. If a vertex v has not yet left the stack for the first time then reach$(v) =$ **false**; otherwise it is **true**.

Mark vector. Prior to v's entering the stack for the first time, mark$(v) =$ **false**; mark(v) is set to **true** when v is stacked. Upon leaving the stack, v is unmarked only if a new cycle was found during its sojourn on the stack. If v leaves the stack with the mark on, then it will be unmarked, when a vertex z_1 is popped from the stack and a new cycle was found during z_1's sojourn on the stack, if there exists a path $z_k, z_{k-1}, \ldots, z_1$ ($z_k = v$) such that $z_{i+1} \in B(z_i)$ ($1 \leqslant i < k$) at that time.

Boolean variables f and g. The combined action of the Boolean variables f and g propagates the information that a new cycle was found with a vertex, say, v at the top of the stack, for all the vertices on the stack which are below v.

Integer variable q. Szwarcfiter and Lauer note that when their algorithm detects a cycle, it is a *new* cycle (i.e., as yet unenumerated) if and only if at least one of its vertices has never been deleted from the stack. In order to detect duplicate cycles a variable q, local to the recursive procedure, is used. For a given computation of this procedure q indicates the topmost vertex of the stack that has never been unstacked. Suppose the arc (v, w) is explored and vertex w is marked. If position(w) $\leqslant q$, a new cycle has been found. Otherwise this is a duplicate cycle, whereupon v is inserted in $B(w)$, and w is deleted from $A(v)$.

This duplicate-cycle detection mechanism is unnecessarily elaborate and does not take advantage of the nature of the backtracking search strategy. The following lemma shows that we can replace this mechanism for identifying new cycles by a very much simpler condition, i.e., the exploration of an arc (v, w) yields a new cycle if mark(w) = **true** and reach(w) = **false**.

LEMMA 1. *For any computation of the algorithm in* [15] *and for any vertex w in the input digraph* $D = (V, E)$, position(w) $\leqslant q$ *if and only if* reach(w) = **false**, *provided w is marked.*

Proof. Assume reach(w) = **false**. Hence the vertex w has not yet been removed from the stack for the first time and, since it is marked, it is currently on the stack. Therefore the topmost vertex on the stack that has not been unstacked for the first time is either w itself or is a vertex above w on the stack. Thus it follows from the definition of q that $q \geqslant$ position(w).

Assume $q \geqslant$ position(w). Let v be the topmost vertex on the stack, which has not yet been unstacked for the first time. Then $q =$ position(v). Since w is marked, if w is not on the stack, position(w) $= n + 1 > q$, so w must be on the stack, and furthermore, since position(v) $\geqslant$ position(w), either $v = w$ or else $v \neq w$ and w is below v on the stack.

If $v = w$ then reach(w) = reach(v) = **false** and the lemma is trivially true. So let $v \neq w$, reach(w) = **true** (i.e., w has been removed from the stack at least once), and v be k places above w on the stack.

If $k = 1$ then v immediately follows w on the stack, so $v \in A(w)$. Therefore, when w was on the stack before, the arc (w, v) must have been explored before vertex w was closed, i.e., before $A(w)$ was completely explored. But v could not have been placed on the stack; otherwise it would have been unstacked before w could have been. The only way that this could happen (i.e., v not placed on the stack) would be if v were marked. Hence v must have been on the stack below

w. But now v is on the stack above w; hence v must have been removed from the stack, which contradicts our hypothesis. So, if $k = 1$, then reach(w) = **false**.

Assume now that reach(w) = **false** whenever $q \geqslant$ position(w) and v is $r < k$ places above w on the stack. Consider x, the vertex immediately above w on the stack. Since x is on the stack, it follows from Lemma 4 in Section 6 that it is marked. Furthermore, since x is marked and not above v on the stack, it follows that $q \geqslant$ position(x). Therefore, since v is $k-1$ places above x on the stack, by our inductive hypothesis reach(x) = **false**. Now by arguing about x the same way as we argued about v when $k = 1$, it follows that reach(w) = **false**. ■

Below is an ALGOL-like formulation of the cycle-finding stage of the algorithm incorporating the simplified duplicate-cycle detection mechanism.

```
procedure CYCLE($v, f$); value $v$; integer $v$; boolean $f$;
begin
  procedure NOCYCLE($x, y$); value $x, y$; integer $x, y$;
  begin
    insert $x$ in $B(y)$;
    delete $y$ from $A(x)$;
  end NOCYCLE;
  procedure UNMARK($y$); value $y$; integer $y$;
  begin
    mark($y$) : = false;
    for $x \in B(y)$ do
    begin
      insert $y$ in $A(x)$;
      if mark($x$) then UNMARK($x$);
    end;
    empty $B(y)$;
  end UNMARK;
  boolean $g$;
  mark($v$) : = true;
  $f$ : = false;
  insert $v$ on the stack;
  for $w \in A(v)$ do
    if $\neg$mark($w$) then
    begin
      CYCLE($v, g$);
      if $g$ then $f$ : = true
          else NOCYCLE($v, w$);
    end
    else if $\neg$reach($w$) then
      begin
        output cycle $w$ to $v$ from stack, then $w$;
        $f$ : = true;
      end
      else NOCYCLE($v, w$);
  delete $v$ from the stack;
  if $f$ then UNMARK($v$);
  reach($v$) : = true;
end CYCLE;
```

The calling sequence for the algorithm is given in the next section once some additional preprocessing has been described.

4. ADDITIONAL PREPROCESSING: BREAKING DOWN SCCS INTO THEIR CONSTITUENT STRONGLY CONNECTED BLOCKS

Consider some arc (x, y) in a digraph D, where the two incident vertices, x and y, are in different sccs. This arc cannot lie on any cycle, since all the vertices on a cycle must be mutually reachable and hence in the same scc. Any exploration of such arcs in search of cycles is therefore a waste of time. Thus, in the Szwarcfiter-Lauer algorithm [15], the digraph D is broken down into its sccs. Only nontrivial sccs are actually searched for cycles.

An scc of a digraph can comprise more than one block (that is, it can contain one or more cutpoints). Yet all the arcs of a cycle must lie in the same block. We can take advantage of this fact by breaking down an scc into its constituent strongly connected blocks and searching these blocks individually for cycles. The benefit to be obtained by doing so is illustrated by the following example.

Consider the way that the Szwarcfiter-Lauer algorithm enumerates the cycles of the digraph in Figure 1. This digraph is strongly connected and comprises two blocks, the vertex $k+4$ being the cutpoint. The Szwarcfiter-Lauer algorithm will select vertex $k+1$ as the start vertex, since it has the largest indegree. The single cycle in the upper block is enumerated the first time that vertex $k+4$ is put on the stack. On each of the subsequent $k-1$ occasions when vertex $k+4$ is stacked, the upper block is reexplored; its remaining $n-k-4$ vertices are deleted from their adjacency lists and then restored to them when vertex $k+4$ is unmarked. (Vertex $k+4$ is always unmarked when it is removed from the stack, since every time it is explored, a *new* cycle is found.) Thus, there are $k-1$ fruitless reexplorations of the upper block. The problem stems from the fact that when the unmarking process is invoked, vertices in blocks other than the one in which the most recently enumerated cycle was found can be restored to the adjacency lists.

Two solutions to this problem would be:

(1) to restrict the unmarking process in such a way that a vertex, in a certain block, deleted from its adjacency list will be restored to it again if and only if a cycle from this block has been enumerated since that vertex's deletion; or (2) to break down the digraph's nontrivial sccs into their constituent blocks and search these for cycles. (Note that the blocks of a strongly connected digraph are all strongly connected; for a proof of this, see Theorem 9.7 in Harary et al. [4].)

In this paper the authors have adopted the latter solution in order to facilitate the preprocessing stage to be described in the next section. Further, in order to incorporate the preprocessing stage discussed herewith, the calling sequence for procedure CYCLE is given below. The method for selecting a start

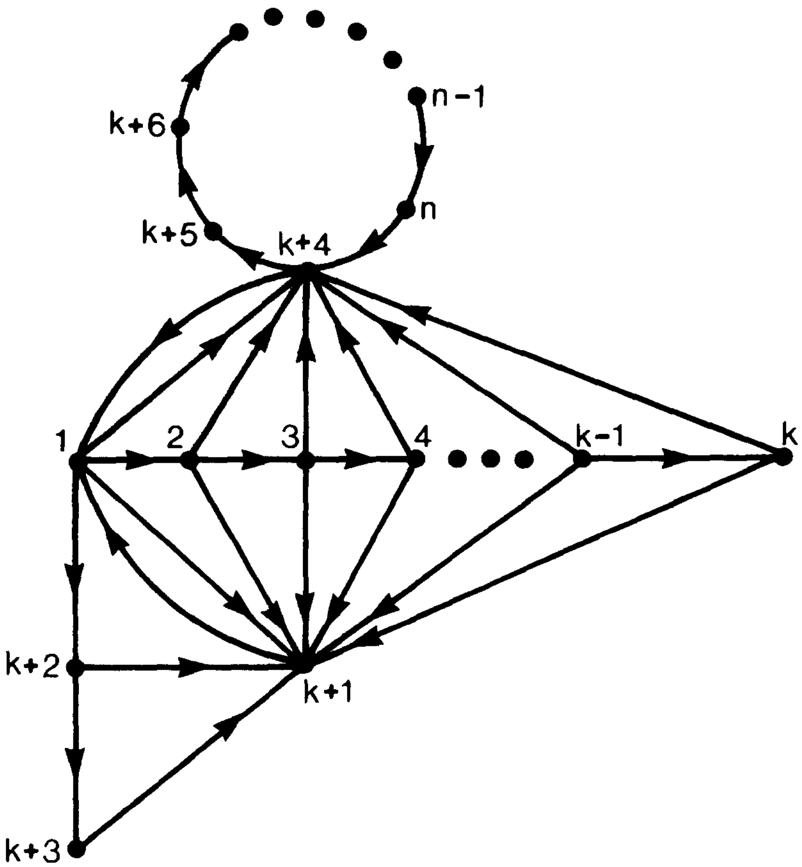

Fig. 1.

vertex—taking the vertex with the lowest ratio of outdegree to indegree—is taken from Hansel et al. [3]. This is better than the method given in [15]—taking the vertex with the highest indegree—on digraphs such as that in Figure 2, since this latter method *could* result in vertex $k+2$ being selected as the start vertex, whereupon vertices $2k+3$ to $3k+3$ are explored k times. However, using the method in [3], they are explored just once, since vertex $3k+3$ is selected as the start vertex. Nevertheless, even this method of selecting a start vertex still has its drawbacks; for example, if the arc $(3k+3,2k+3)$ is added to the digraph in Figure 2, then $k+2$ *is* chosen as the start vertex.

```
begin
  comment finding the cycles of a digraph;
  read in the digraph $D$;
  $A$ : = adjacency lists of the sccs of $D$;
  for each nontrivial scc do
```

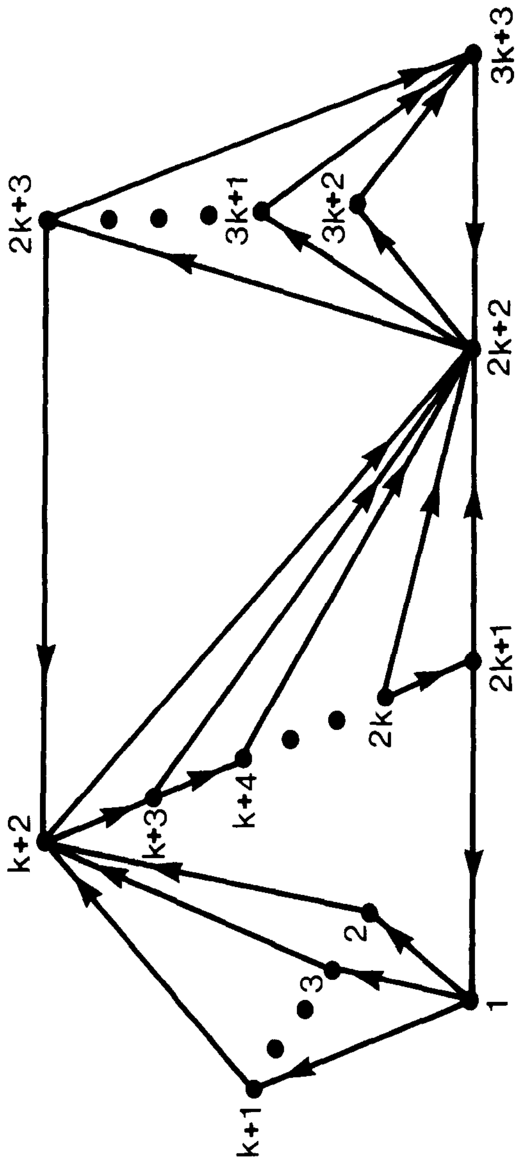

Fig. 2.

```
  begin
    for each block in the current scc do
    begin
      for each vertex $j$ in the current block do
      mark($j$): = reach($j$): = false;
      comment select a start vertex;
      $s$: = the vertex with the lowest ratio of outdegree to indegree in the current block;
      CYCLE($s$, dummy);
    end
  end
end
```

In [16] there is described an algorithm, with $O(e)$ time complexity, which can be used to find the blocks of an scc. In order to use this algorithm, it is necessary to create an undirected version of the input digraph, marking any added vertices in the adjacency lists for future identification. Procedure CYCLE must then be altered to make it ignore such added vertices. This undirected version of each scc is also used in the preprocessing stage to be described in the next section.

5. ADDITIONAL PREPROCESSING: BREAKING STRONGLY CONNECTED BLOCKS DOWN INTO THEIR CONSTITUENT STRONG TRICONNECTED COMPONENTS

Suppose some digraph D has been read in and divided up into strongly connected blocks by the preprocessing stages already described. Consider one such strongly connected block $D_b = (V_b, E_b)$. Suppose that D_b is not triconnected; thus D_b contains at least one separation pair, say, $\{v, w\}$. (Hopcroft and Tarjan [5] have published an algorithm, with time complexity $O(n+e)$, for finding the separation pairs of a biconnected graph.)

Let E_1 be a separation class with respect to $\{v, w\}$. Let $E_2 = E_b - E_1$. If there is an arc from v to w and/or an arc from w to v, these can be arbitrarily assigned to E_1 or E_2. E_1 and E_2 can be considered to be the arc sets of two digraphs, $D_1 = (V_1, E_1)$ and $D_2 = (V_2, E_2)$, where V_1 comprises every vertex incident to an arc in E_1 and V_2 comprises every vertex incident to an arc in E_2. We note that $V_1 \cap V_2 = \{v, w\}$ and $E_1 \cap E_2 = \varnothing$, and also that there must be at least one vertex in V_1 and in V_2 other than v and w, and furthermore that any strongly connected block with less than four vertices must be triconnected.

Let the set of cycles in D_b be denoted by C_b. Furthermore, let $\{C_1, C_2, C_3, C_4\}$ be a partition of C_b defined as follows:

C_1 is the set of cycles whose arcs are exclusively in D_1.
C_2 is the set of cycles whose arcs are exclusively in D_2.
C_3 is the set of cycles each comprising a v-w path in D_1 and a w-v path in D_2.
C_4 is the set of cycles each comprising a w-v path in D_1 and a v-w path in D_2.

It is proposed that the cycles of D_b be enumerated in the following way:

1. Enumerate the cycles in C_1.
2. Enumerate the cycles in C_2.
3. Find the set of v-w paths in D_1, storing it in list VWPATHS.
4. Enumerate the cycles belonging to set C_3 as follows: Search D_2 for w-v paths. Whenever such a path is found, concatenate it with each path in VWPATHS in turn; the resulting sequence of vertices represents a cycle in C_3.
5. Find the set of w-v paths in D_1, storing it in list WVPATHS.
6. Enumerate the cycles belonging to set C_4 as follows: Search D_2 for v-w paths. Whenever such a path is found, concatenate it with each path in WVPATHS in turn; the resulting sequence of vertices represents a cycle in C_4.

When enumerating the cycles in C_1 and C_2, the subdigraphs D_1 and D_2 are treated just like any other digraph input to the algorithm. That is, they are broken down into sccs, the nontrivial sccs are broken down further into strongly connected blocks, these blocks are searched for separation pairs, and so on. The cycle-finding procedure is only called when a triconnected component has been found.

Dinh et al. [2] have published an algorithm for finding the paths between two given vertices in a digraph. This algorithm has time complexity $O((n+e)(p+1))$, where p is the number of paths in the digraph. (The performance of this algorithm could be improved by incorporating into it the marking system given in [14] or [15].) An alternative to this algorithm is the one published in [7].

Consider now the way in which the original formulation [15] of the Szwarcfiter-Lauer algorithm handles the digraph D, in Figure 3. Let $D=(V,E)$ have n vertices ($n>4$) and e arcs, where n is assumed to be even. Let $k=n/2$; D comprises two complete subdigraphs, D_x and D_y, both with k vertices and connected as in Figure 3. Suppose, without loss of generality, that a_1 is selected as the start vertex. If the algorithm starts to search subdigraph D_x, then whenever a_2 is stacked, the whole of subdigraph D_y is reexplored. Within subdigraph D_x there are $q=1+\sum_{i=1}^{k-2}P(k-2,i)$ paths from a_1 to a_2. Thus, D_y

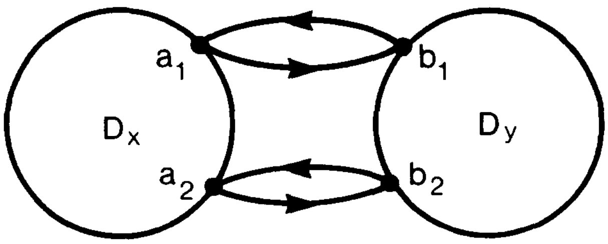

Fig. 3.

will be explored q times. [$P(n,r)$ denotes the number of ways of arranging r of n distinct objects.]

Once the last arc leading from a_1 into D_x has been explored, the arc (a_1, b_1) will be stacked next. Then subdigraph D_y will be completely explored yet again. Moreover, there are also q paths from b_1 to b_2, so, on the subsequent q occasions when b_2 gets stacked, subdigraph D_x will be completely reexplored.

If the algorithm proposed in the current section is used in place of the one in [15], subdigraphs D_x and D_y are both searched exactly three times: once to find the cycles whose arcs all lie in the same subdigraph, once to find the v-w paths, and once to find the w-v paths.

The proposed strategy amounts to a way of substituting space for time. The price to be paid for the benefits described above is that the space complexity of the proposed algorithm can be exponential in the size of the digraph, since it depends on the number of paths as well as the number of arcs and vertices in the digraph.

Hereinafter we present an ALGOL-like formulation of the cycle-finding algorithm.

Procedure FINDCYCLES (D) selects a start vertex and then calls the procedure CYCLE described in Section 3 (adjusted so as to ignore any arcs that were created to form D's undirected version). Note that the user could substitute any cycle-finding algorithm he chooses into procedure CYCLE.

Procedure FINDPATHS(D, s, t,pathlist) is based on the path generation algorithm given in [2] or, alternatively, on that in [7]. D is the digraph to be searched, s is the start vertex, t is the target vertex, and pathlist is for the storage of the set of s-t paths in D.

Procedure ENUMERATECYCLES(D, t, s,pathlist) is a modified version of FINDPATHS. Having found a path from t to s in D, it concatenates it with every path in pathlist in turn and outputs the results as cycles.

Procedure FINDPAIR(D, D_1, D_2, v, w) is based on the algorithm of Hopcroft and Tarjan [5] for finding the triconnected components of a graph. It takes a digraph D as input and, using the undirected version of D, attempts to find a separation pair. If there is such a pair, the associated vertex numbers are assigned to v and w. Then a separation class (with respect to $\{v, w\}$) E_1 of arcs is obtained, and a digraph $D_1 = (V_1, E_1)$ is formed such that V_1 is the set of vertices incident to arcs in E_1. Another digraph, $D_2 = (V_2, E_2)$, is also formed such that $E_2 = E - E_1$ and V_2 is the set of vertices incident to arcs in E_2. D_1, D_2, v, and w are all returned to the procedure that calls FINDPAIR.

```
begin
  comment finding the cycles of a digraph;
  procedure FIND($D$);
  begin
    for each nontrivial scc $D_s$ of $D$ do
α:  for each block $D_b$ in $D_s$ do
```

```
        begin
          FINDPAIR($D_b$, $D_1$, $D_2$, $v$, $w$);
          if there are no separation pairs in $D_b$
          then FINDCYCLES($D_b$)
          else
          begin
            FIND($D_1$);
            FIND($D_2$);
            FINDPATHS($D_1$, $v$, $w$, VWPATHS);
            if VWPATHS $\neq \varnothing$
            then ENUMERATECYCLES($D_2$, $w$, $v$, VWPATHS);
            FINDPATHS($D_1$, $w$, $v$, WVPATHS);
            if WVPATHS $\neq \varnothing$
            then ENUMERATECYCLES($D_2$, $v$, $w$, WVPATHS);
          end
        end
      end FIND;
      read in the digraph $D$;
      FIND($D$);
    end
```

Readers should note that there are better methods of implementing the idea of breaking the digraph down into its triconnected components than the method described above, which is just the simplest to describe. For example, one could use a modified version of the cycle-finding algorithm which enumerates paths at the same time as it enumerates cycles (rather than reexploring each triconnected component afresh). Another improvement on the basic idea would be to have the algorithm make an intelligent selection of a separation pair when there is a choice. For example, the reader might like to consider the different ways in which the digraph in Figure 2 can be divided up according to which separation pair is selected by the algorithm. Several selection criteria suggest themselves:

(1) Attempt to select a separation pair $\{v,w\}$ such that in one of the ensuing triconnected components there are v-w paths but no w-v paths. In Figure 2 this happens when the digraph is divided up using the separation pair $\{k+2,2k+2\}$.

(2) Select the separation pair which divides up the digraph into the most similar-sized triconnected components.

(3) Assume that some pair of vertices $\{v,w\}$ has already been used as a separation pair to divide a digraph D into two triconnected components, D_1 and D_2, and that, when D_1 is passed to procedure FINDPAIR, $\{v,w\}$ is found to be a separation pair in D_1 also. Then it is advantageous to select the same pair again, since the algorithm kills two birds with one stone, when finding the paths between v and w.

6. CORRECTNESS PROOF

Let $D=(V,E)$ be a triconnected digraph which is passed to procedure FINDCYCLES. Lemmas 2, 3, and 4 below are taken from [14].

LEMMA 2. *Let $v_1, v_2, \dots, v_k, v_1$ be a cycle such that $v_1, v_2, \dots, v_k$ or a cyclic permutation of it has already appeared in the top k positions of the stack at some earlier time, and at least one of these vertices has been deleted from it before. If $v_1, v_2, \dots, v_k$ now occupy the top k positions of the stack, then all $v_1, v_2, \dots, v_k$ have already been deleted from it.*

LEMMA 3. *Let $v_1, v_2, \dots, v_k, v_1$ be a convenient cyclic permutation for a cycle, such that v_1 was the first among the v_j $(1 \leq j \leq k)$ to ever enter the stack. Then there exists a configuration of the stack such that before v_1 leaves the stack for the first time, $v_1 v_2 \dots v_j$ $(1 \leq j \leq k)$ appear in the top j positions of the stack.*

LEMMA 4. *If a vertex is on the stack, then it is marked.*

LEMMA 5. *Each cycle of D is listed at least once.*

Proof. Consider the set $\mathbb{C}$ of cycles in D. Let each cycle be represented as a sequence of vertices, with the vertex which entered the stack before any of the others in the sequence being the first term as well as the last term in the sequence. Suppose $C \in \mathbb{C}$, and let $C = v_1, v_2, \dots, v_k, v_1$. By Lemma 3 $v_1 v_2 \dots v_k$ will eventually occupy the top k positions of the stack before v_1 leaves the stack for the first time. Thus, at the time that $v_1 v_2 \dots v_k$ occupy the top k positions on the stack and the arc (v_k, v_1) is about to be explored, reach(v_1) = **false**, and, by virtue of Lemma 4, mark(v_1) = **true**. Thus the cycle C is enumerated. ■

LEMMA 6. *Each cycle of D is listed at most once.*

Proof. Suppose that at some point in the computation, the top k stack entries are $v_1 v_2 \dots v_{k-1} v_k$, and that the arc (v_k, v_1) is about to be explored. Suppose, also, that at some earlier point in the computation the cycle $C = v_1, v_2, \dots, v_k, v_1$ was enumerated. Consider the following two cases:

(a) When C was enumerated, the sequence on the top of the stack was $v_1 v_2 \dots v_k$ and the arc (v_k, v_1) was explored.
(b) When C was enumerated, the sequence on the top of the stack was $v_i v_{i+1} \dots v_{k-1} v_k v_1 v_2 \dots v_{i-1}$ $(1 < i \leq k)$ and the arc (v_{i-1}, v_i) was explored.

In case (a), it is obvious that the arc (v_k, v_1) can be reexplored if and only if v_k has been unstacked and restacked since the last time the arc (v_k, v_1) was explored. In case (b), vertex v_1 is currently *below* the vertices v_j $(2 \leq j \leq k)$, whereas when C was enumerated it was above the vertices v_l $(i \leq l \leq k)$. Thus vertex v_1 must have been unstacked and restacked since C was enumerated.
By Lemma 2 we conclude that reach(v_i) = **true** when $i = 1$ [case (a)] or $1 < i \leq k$ [case (b)], so C will not be reenumerated. ■

LEMMA 7. *Procedure* FIND *enumerates the cycles of a triconnected digraph correctly.*

Proof. This follows directly from Lemmas 5 and 6. ■

LEMMA 8. *Procedure* FIND *will not enumerate any cycles in an acyclic digraph.*

Proof. An acyclic digraph does not contain any nontrivial sccs. Thus the statement starting on the line labeled α in procedure FIND will not be executed if the digraph passed to FIND as an actual parameter is acyclic. ■

REMARK. The action of the algorithm described in Section 5 of this paper—finding a digraph's sccs, breaking these down into their constituent strongly connected blocks, dividing these blocks in two using a separation pair, and so on—can be regarded as constructing a tree, where the root node is the input digraph and each other node is a subdigraph of its parent node, generated by one of the processes described above. The subdigraphs represented by the leaf nodes of this tree will either be triconnected or acyclic components. Lemmas 7 and 8, respectively, show that these categories of digraph are dealt with correctly by the algorithm.

LEMMA 9. *Procedure* FIND *correctly enumerates the cycles in a cyclic digraph that is not triconnected.*

Proof. Procedure FIND divides the digraph into sccs and searches each nontrivial scc for cycles separately. This is justified by the fact that any two vertices lying on a common cycle must be mutually reachable and thus in the same nontrivial scc.

It is obvious that all the vertices and arcs in any cycle must lie in the same block. Thus procedure FIND is justified in breaking each nontrivial scc down into its constituent strongly connected blocks, and searching each of these for cycles separately.

It remains to be proven that the algorithm correctly enumerates the cycles in a strongly connected block that is not triconnected.

Let $D_b = (V_b, E_b)$ be a strongly connected block which is not triconnected. Suppose that D_b is some internal node in the tree, described in the Remark above, which is generated for some input digraph D. Regardless of whether D_b is passed to procedure FIND or found to be an scc (say, D_s), D_b will be passed as an actual parameter to procedure FINDPAIR. This latter procedure will discover a separation pair $\{v, w\}$, and will return two subdigraphs of D_b, namely $D_1 = (V_1, E_1)$ and $D_2 = (V_2, E_2)$, in the manner described in Section 5.

Consider the two subdigraphs D_1 and D_2. Let the sets of cycles in these two subdigraphs be denoted by C_1 and C_2, respectively. Both these subdigraphs will be passed as actual parameters in recursive calls to procedure FIND. If either subdigraph is triconnected, then by Lemma 7 its cycles will be correctly enumerated. If either subdigraph is acyclic, then by Lemma 8 no cycles will be inadvertently enumerated while it is being examined. If either subdigraph is cyclic but not triconnected, then the proof that its cycles are correctly enumerated is the same as the proof that D_b's cycles are correctly enumerated—the purpose of the present lemma.

Let C_b denote the set of cycles in D_b. Obviously, C_1 and C_2 are disjoint subsets of C_b. Let $C_r = C_b - C_1 - C_2$. Any cycle in C_r must have at least one arc, say e_1, in D_1 and at least one arc, say e_2, in D_2. These two arcs must be connected by two paths: p_1, which goes from e_1 to e_2, and p_2, which goes from e_2 to e_1. By the definition of a cycle these two paths can have no vertices in common.

Consider path p_1. It starts in D_1 and ends in D_2. Thus one of the vertices in path p_1 must be either v or w. Suppose, without loss of generality, that vertex w lies on path p_1. Then vertex w cannot lie on path p_2, yet path p_2 must somehow run from D_2 to D_1. Since the only vertex besides w which connects D_2 to D_1 is v, it follows that vertex v lies on path p_2. Thus, any cycle in C_r comprises a v-w (w-v) path in D_1 concatenated with a w-v (v-w) path in D_2.

Let the sets of v-w and w-v paths in D_1 be denoted by VW_1 and WV_1, respectively. Define similarly the sets WV_2 and VW_2 in D_2. The set C_r can be regarded as comprising two disjoint subsets, C_3 and C_4, where $C_3 = VW_1 \oplus WV_2$ and $C_4 = WV_1 \oplus VW_2$. In the statement of the algorithm in Section 5 the set VW_1 will be enumerated and assigned to VWPATHS by the call to FINDPATHS, and the set C_3 will be enumerated when the set VW_1 is passed as an actual parameter to ENUMERATECYCLES. The set C_4 will be enumerated in a similar way in subsequent calls to FINDPATHS and ENUMERATECYCLES. ■

THEOREM 1. *The proposed algorithm for finding the cycles of a digraph is correct.*

Proof. This follows immediately from Lemmas 7, 8, and 9. ■

7. CONCLUDING REMARKS

None of the ideas presented in this paper have any effect on the worst-case time complexity of the algorithm in [15], which is $O(n+e)$ per cycle. The methods described herein for breaking down the given digraph into simpler subdigraphs can, however, lead to improvements in execution time. Another reason for dividing up the problem under consideration into *subproblems* might be to increase the scope for parallel processing.

Further preprocessing along these lines, e.g. looking for a minimal separation tuple rather than just a separation pair, becomes unattractive quite quickly unless n is very large, since the number of different sets of paths that need to be enumerated rapidly increases.

The modified Szwarcfiter-Lauer algorithm together with the preprocessing phase suggested in Section 4 has already been implemented in ALGOL 60. The full algorithm presented in Section 5 is being implemented, and the results will appear in a forthcoming publication.

Finally, in response to the question posed in [15] concerning "the existence of an algorithm that would find all elementary cycles of a digraph, in such a way that any edge or vertex would be unsuccessfully explored, at most a constant number of times, during the entire process," we refer the reader to Valiant [18, p. 417], who classifies the problem of enumerating the cycles of a digraph as *#P-complete*. #P-complete problems are a class of computationally equivalent enumeration problems that are at least as difficult as the NP-complete problems in the sense that even if P = NP, the most efficient possible cycle enumeration algorithm may still have to make a number of unsuccessful searches of the input digraph's arcs that is related to the number of cycles in the digraph.

REFERENCES

1. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.
2. T. Dinh, S. Tsukiyama, I. Shirakawa, and H. Ozaki, An algorithm for generating all the directed paths and its application, *Inform. Process. Japan* (Joho-Shori) 16:774–780 (1975).
3. K. Hansel, W. Jänicke, and Ju. M. Wolin, Zur bestimmung aller elementarkreise in gerichten graphen, *Math. Operationsforsch. Statist. Ser. Optim.* 9:427–440 (1978).
4. F. Harary, R. Z. Norman, and D. Cartwright, *Structural Models: An Introduction to the Theory of Directed Graphs*, Wiley, 1965.
5. J. E. Hopcroft and R. E. Tarjan, Dividing a graph into triconnected components, *SIAM J. Comput.* 2:135–158 (1973).
6. P. Mateti and N. Deo, On algorithms for enumerating all circuits of a graph, *SIAM J. Comput.* 5:90–99 (1976).
7. S. Niculescu, An algorithm to find the paths between two vertices of a graph (in Romanian), *Stud. Cerc. Mat.* 31:581–592 (1979).
8. N. Pippenger, A time-space trade-off, *J. Assoc. Comput. Mach.* 25:509–515 (1978).
9. R. C. Read and R. E. Tarjan, Bounds on backtrack algorithms for listing cycles, paths, and spanning trees, *Networks* 5:237–252 (1975).
10. P. K. Srimani, Generation of directed circuits in a directed graph, *Proc. IEEE* 67:1361–1362 (1979).
11. P. K. Srimani and A. Sengupta, Algebraic determination of circuits in a directed graph, *Internat. J. Systems Sci.* 10:1409–1413 (1979).
12. M. M. Syslo, Algorithm 459: The elementary circuits of a graph, *Comm. ACM* 16:632–633 (1973).
13. M. M. Syslo, Remark on algorithm 459: The elementary circuits of a graph, *Comm. ACM* 18:119 (1975).
14. J. L. Szwarcfiter and P. E. Lauer, A new backtracking search strategy for the enumeration of the elementary cycles of a directed graph, Univ. of Newcastle upon Tyne, Technical Report Series, No. 69, 1975.

15. J. L. Szwarcfiter and P. E. Lauer, A search strategy for the elementary cycles of a directed graph, *BIT* 16:192–204 (1976).
16. R. Tarjan, Depth-first search and linear graph algorithms, *SIAM J. Comput.* 1:146–160 (1972).
17. S. Tsukiyama, I. Shirakawa, and H. Ozaki, An algorithm for generating the cycles of a digraph, *Electron. Comm. Japan* 58:8–15 (1975).
18. L. G. Valiant, The complexity of enumeration and reliability problems, *SIAM J. Comput.* 8:410–421 (1979).
19. H. Weinblatt, A new search algorithm for finding the simple cycles of a finite directed graph, *J. Assoc. Comput. Mach.* 19:43–56 (1972).

Received March 1982