A New Search Algorithm for Finding the Simple Cycles of a Finite Directed Graph

HERBERT WEINBLATT*

Bell Telephone Laboratories, Holmdel, New Jersey

ABSTRACT. In many applications of directed graph theory, it is desired to obtain a list of the simple cycles of the graph. In this paper, a new search algorithm for finding the simple cycles of any finite directed graph is presented, and the validity of the algorithm is proven. The algorithm has been implemented experimentally in Snobol3, and tests indicate that the algorithm is reasonably fast. (The simple cycles of a 193 vertex graph were obtained in 6.8 seconds on an IBM 7094 computer.)

KEY WORDS AND PHRASES: directed graphs, cycles, path examination, feedback paths, program segmentation, flow-chart analysis, algorithms, search algorithms, Snobol

CR CATEGORIES: 5.32

1. Introduction

A directed graph may be described as a set of points (*vertices*) together with a set of directed line segments (*arcs*) such that each arc connects precisely two vertices, "originating" on one vertex and "terminating" on the other. A *path* connecting one vertex, v_0, to another, v_n, is an ordered collection of arcs $a_1a_2 \cdots a_n$ such that a_1 originates on v_0, each of the other arcs originates on the vertex on which the preceding arc terminates, and a_n terminates on v_n. In order to show explicitly which vertices are encountered by a path, it is also possible to include the vertices in the path; thus $v_0a_1v_1a_2v_2 \cdots v_{n-1}a_nv_n = a_1a_2 \cdots a_n$.

A path is *simple* if it encounters no vertex twice, and *cyclic* if it originates and terminates on the same vertex. The arcs and vertices of a cyclic path without regard to its endpoints form a *cycle* (or, in programming terminology, a "loop"). Thus, the two cyclic paths $v_1a_2v_2a_1v_1$ and $v_2a_1v_1a_2v_2$ both correspond to the same cycle. Cycles may be represented by using any representation of any of the corresponding cyclic paths.

A cyclic path is *simple-cyclic* if it encounters one vertex twice (the one on which it originates and terminates), and no other vertex more than once. A cycle is *simple* if it corresponds to a simple-cyclic path. Clearly, every cycle is composed of one or more simple cycles. In the remainder of this paper the term "cycle," when not otherwise modified, will be used only to refer to simple cycles.

In many applications of directed graph theory, it is desired to obtain a list of the cycles of the graph. This information can then be used: (a) to help break the feed-

* Present address: Division of Engineering, Brown University, Providence, RI 02912.

back paths (of a control system, a logical network, a computer program, etc.) [9]; (b) as part of a sophisticated system for optimizing computer programs [1]; or (c) to perform more general types of analysis on computer programs, such as estimation of running times [6].

This paper presents a new search algorithm which finds all the simple cycles of a directed graph and lists each one once. The algorithm investigates the paths of a graph in such a way that each individual arc of the graph is examined once and only once. Cycles are discovered either when the path being followed is found to be cyclic or by combining parts of previously discovered cycles with a part of the path currently being processed. The process of combining parts of cycles makes it necessary to save representations of all simple cycles and, sometimes, to examine parts of these cycles (but not their individual arcs) a number of times. Hence, the algorithm has relatively large storage requirements; however, it does appear to be quite efficient from the standpoint of running time.[1]

The algorithm has been experimentally implemented in Snobol3, and the speed and storage requirements of this implementation are examined in some detail in Section 5.

2. *The Algorithm*

2.1 AN INFORMAL DESCRIPTION. The algorithm begins processing the graph by removing from the graph any vertices on which no arcs terminate, and also removing any arcs originating on these vertices. This step is applied repeatedly until no such vertices are left. Then, in a similar fashion, all vertices on which no arc originates are removed, and also any arcs terminating on these vertices. This step is also applied repeatedly until no such vertices are left. The effect of these two processes is to reduce the size of the graph by eliminating vertices which cannot belong to any cycles.

The algorithm next selects one of the remaining vertices as a "starting point," and begins to examine the paths which emanate from the vertex. Each path is explored until a vertex is encountered which has already been examined. When such a vertex is encountered, the algorithm backs up along the path to the last branch point[2] which has been encountered and for which theire remains at least one arc which originates on the vertex and which has not yet been explored. Choosing such an arc, the algorithm then sets off through the graph in a new direction. If no such vertex exists then, if possible, a vertex which has not yet been encountered is

[1] The algorithm is related to one reported by Tiernan [12] after this article was originally submitted for publication, but, in an attempt to improve computational efficiency, performs substantially more bookkeeping. Tiernan claims that, since his algorithm considers each cycle only once, it is the "theoretically most efficient search algorithm." However, he does not consider improving efficiency in any other way, such as minimizing the number of examinations of each individual arc (as is done by the algorithm presented in this article). Although the present algorithm clearly requires more storage than does Tiernan's, it also appears to be substantially faster. Indeed, although Tiernan indicates that his algorithm would probably be impractically slow for graphs with average density and more than 100 arcs, an experimental Snobol3 implementation of the present algorithm on an IBM 7094 has processed a 294-arc graph in less than seven seconds.

[2] A *branch point* is a vertex on which two or more arcs originate (i.e. a vertex whose "outdegree" is greater than one.)

selected as a new starting point, and the algorithm proceeds to explore the paths which emanate from this vertex. If all vertices of the graph have been examined, the algorithm is terminated.

As the algorithm proceeds through the graph, it keeps track of where it has been by placing the "elements" (i.e. the arcs and vertices) which it has encountered on a list called the TT ("trial thread"). The TT always represents a path through the graph from the starting point to the element which the algorithm is currently examining. When the algorithm backs up along a path, it removes from the TT all the arcs and vertices through which it must pass during this backing up.

When the algorithm encounters a vertex which has been previously examined, before backing up, it checks whether any combination of arcs which it has so far examined form a cycle which has not already been discovered. If the vertex is still on the TT precisely one such cycle will exist: it will consist of the vertex together with the "tail" of the TT with respect to this vertex.

If the vertex is no longer on the TT, such cycles can exist only if one or more cycles containing the vertex have previously been found. If this is the case, then a recursive precedure is entered which attempts to find paths which originate at the re-encountered vertex and terminate at a vertex which is still on the TT. It will be shown in Section 4.2 that any such path is either a terminal subpath of a previously discovered cycle, or is decomposable, in one or more ways, into a set of such terminal subpaths.[3] Hence, only terminal subpaths of previously discovered cycles need be considered by this procedure. For each path found by the procedure, a new cycle is formed by concatenating the terminal subpath of the TT with the path. (For example, suppose cycles $C_1 = v_1a_2v_2a_1v_1$ and $C_2 = v_2a_3v_3a_4v_2$ have already been found, and, with the $TT = v_1a_5$, the algorithm re-encounters v_3. Then, by concatenating subpath v_1a_5 of the TT, subpath $v_3a_4v_2$ of C_2, and subpath a_1v_1 of C_1, a new cycle $v_1a_5v_3a_4v_2a_1v_1$ is found.)

2.2 DEFINITIONS AND EXPLANATIONS. During the entire application of the algorithm, the TT represents a path from some starting vertex to the arc or vertex currently being examined. (Strictly speaking, a path always originates on a vertex and terminates on a vertex. However, for the purposes of this discussion and also that of Section 4, it is convenient to use the term "path" to refer to those strings which are obtained from the representation of a true path by deleting initial and/or terminal vertices.)

The representation of paths used in the formal statement of the algorithm (Section 2.3) and in the validity proof (Section 4) shows both arcs and vertices explicitly. (Two alternative representations will be discussed in Section 2.4.)

The vertex on which an arc a terminates will be represented by $T(a)$, and the vertex on which a path P terminates (or "ends") will be represented by $End(P)$.

The *state* of an arc $S(a)$, is a two-valued function; the values of this function, 0 and 2, indicate respectively that the arc has never been on the TT, or that the arc has been (and may still be) on the TT. Similarly, the state of a vertex, $S(v)$, is a three-valued function; the values of this function, 0, 1, and 2, indicate respectively that the vertex has never been on the TT, is now on the TT, or has been on the TT but has since been removed. [Arcs are never examined while they are on the TT;

[3] Cycles are represented as cyclic paths which originate on the first vertex of the cycle to have been placed on the TT. A "terminal subpath of a cycle" refers to a subpath which is terminal with respect to the cyclic path which is used to represent a cycle.

hence this state, $S(a) = 1$, is of no interest and has been combined with the final state, $S(a) = 2$.]

The *first vertex of a cycle* will be that one of the cycle's vertices which is first encountered during a particular application of the algorithm. Each cycle which is discovered by the algorithm will be represented by a cyclic path having the first vertex of the cycle as its initial and terminal vertex. A "list of cyclic paths" will be used to keep track of the cycles which have been discovered.

Cycles may be discovered by Step ($E3$) of the algorithm, or by the recursive subroutine "*Concat*." If a cycle is discovered by *Concat* and one or more recursive calls to *Concat* have occurred since the last external call, then in order to determine whether or not the cycle has already been found the cyclic path representing the cycle must be compared with the cyclic paths representing any other cycles which have been discovered since the last external call to *Concat*. (It will be shown in Section 4.4 that this is the only time that checking for such duplication is necessary.) To make it possible to determine easily whether a given execution of *Concat* is a top-level or a recursive execution, the variable "*Recur*" is used as a switch: *Recur* is reset to 0 whenever *Concat* is called externally, and set to 1 whenever it is called recursively.

The *tail of a simple path* P with respect to a vertex v belonging to P, $Tail(P, v)$, is obtained from P by taking its terminal subpath originating on v and then deleting v from this representation. [For example, if $P = v_0a_1v_1a_2v_2$, then $Tail(P, v_1) = a_2v_2$.] The *tail of a simple-cyclic path* C with respect to some vertex v of the path is obtained from C by deleting the initial vertex of C and then taking the tail of the resulting path. [For example, if $C = v_2a_1v_1a_2v_2$, then $Tail(C, v_1) = a_2v_2$ and $Tail(C, v_2) =$ void.] The *tail of a cycle* is equivalent to the tail of that cyclic path which is being used to represent the cycle.

The argument of the subroutine *Concat* is a path P which the algorithm is trying to extend, if possible, into as many cyclic paths as can be found. During each call, *Concat* must consider the possibility of concatenating to P the tail [with respect to $End(P)$] of each previously discovered cyclic path containing $End(P)$. These tails are referred to as "cycle-tails." Since the tails of two or more cyclic paths with respect to a given vertex which they contain may be identical, during each execution of *Concat* a list is maintained of the cycle-tails which have been examined so as to avoid duplication of effort.

In the following, the path obtained by concatenating two paths, P and Q, will be represented as P_Q.

2.3 A FORMAL STATEMENT. A formal statement of the algorithm for finding cycles follows.

(A) *Eliminate:*

(1) If there are any vertices on which no arcs terminate, eliminate all such vertices as well as any arcs originating on them, and repeat Step (1); if none, then continue to Step (2).

(2) If there are any vertices on which no arcs originate, eliminate all such vertices as well as any arcs terminating on them, and repeat Step (2); if none, then continue to *Initialize*.

(B) *Initialize:*

For each vertex v set $S(v) := 0$.
For each arc a set $S(a) := 0$.
$TT :=$ *void*.

(*C*) *Select* (a starting vertex):
  (1) Find a vertex v such that $S(v) = 0$;
  if none then stop.
  (2) $TT := v$.
  $S(v) := 1$.
(*D*) *Extend* (the TT).
  (1) Find an arc, a, which originates on $End(TT)$ and such that $S(a) = 0$;
  if none then set $S(End(TT)) := 2$, remove $End(TT)$ from the TT, and go to *Backup*.
  (2) $TT := TT\_a$.
  $S(a) := 2$.
(*E*) *Examine* (the terminal vertex of a):
  (1) $v := T(a)$.
  (2) If $S(v) = 0$ then:
    (a) $TT := TT\_v$;
    (b) $S(v) := 1$;
    (c) Go to *Extend*.
  (3) If $S(v) = 1$, then add $v\_Tail(TT, v)\_v$ to the list of cyclic paths, and go to *Backup*.
  (4) (Case $S(v) = 2$.)
    (a) $Recur := 0$.
    (b) Call $Concat(v)$.
    (c) Go to *Backup*.
(*F*) *Backup:*
  (1) Remove the last arc from the TT.
  (2) If the TT is empty, then go to *Select;* else go to *Extend*.
(*G*) Subroutine $Concat(P)$:
  (1) Establish an empty local list of examined cycle-tails.
  $v := End(P)$.
  (2) Do through Step (10) for each cyclic path CP which is on the list of cyclic paths and which contains v.
  (3) If the cycle-tail $Tail(CP, v)$ is void or it is on the list of examined cycle-tails then go to Step (10).
  (4) Add the cycle-tail to the list of examined cycle-tails.
  (5) If the cycle-tail has any vertices on P, then go to Step (10).
  (6) If $S(End(CP)) = 2$, then
    (a) $Recur := 1$.
    (b) Call $Concat(P\_Tail(CP, v))$.
    (c) Go to Step (10).
  (7) (Case: $S(End(CP)) = 1$.)
  $C := End(CP)\_Tail(TT, End(CP))\_P\_Tail(CP, End(P))$
  (8) If $Recur = 0$, then add C to the list of examined cyclic paths, and go to Step 10.
  (9) If C is not among those cyclic paths which have been added to the list of cyclic paths generated since the last external call to *Concat* (i.e., the last call from *Examine*), then add C to the list of cyclic paths.
  (10) Continue.

2.4 ALTERNATE REPRESENTATIONS OF PATHS. The statement of the algorithm given above requires that representations of the TT and the cycles show explicitly both the arcs and the vertices which belong to a particular path. This requirement is useful for purposes of exposition, but is not actually necessary and has the unfortunate effect of doubling the lengths of the representations of all paths.

It has already been mentioned that a path can be completely defined by listing only the arcs of the path. Hence, the algorithm can be restated in a form which uses a "vertexless" representation of all paths. This restatement requires only the following changes[4]:

[4] For such a representation, the call to *Concat* in Step (*E4*(*b*)) is essentially unchanged; i.e. the argument of the call represents a path of length 0 having terminal vertex v.

(a) Delete "$TT := v$" from Step ($C2$).
(b) Delete "remove $End(TT)$ from TT" from Step ($D1$).
(c) Delete Step ($E2(a)$) ("$TT := TT\_v$").

Another possible representation of paths is one in which only the vertices encountered by a path are shown explicitly. This representation is useful in those applications (such as programming or control systems) in which it is more natural to assign names to the vertices of the graph than to the arcs. Such an "arcless" representation of paths is unambiguous only for graphs which contain no strictly parallel arcs[5]; however, if each set of strictly parallel arcs is replaced by a single arc, and the cycles of the resulting graph are found, the cycles of the original graph can then be determined by substitution. The algorithm can be restated so as to make use of an "arcless" representation of paths by making the following changes:

(a) Delete "$TT := TT\_a$" from Step ($D2$).
(b) Delete Step ($F1$).

An arcless representation of paths was used in the Snobol3 implementation of this algorithm, and, for purpose of conciseness, such a representation will be used in the example of Section 3. However, in the proofs of Section 4, the representations of all paths will show both arcs and vertices explicitly. It should be apparent that all three forms of the algorithm are equivalent for graphs containing no strictly parallel arcs, and that for all graphs the form using the vertexless representation of paths is equivalent to the one in which both the arcs and vertices of a path are represented.

3. *An Example*

For an example of the application of the algorithm, consider the directed graph of Figure 1. The application of the algorithm to this graph is outlined in Figure 2. The leftmost column of Figure 2 enumerates the steps of the algorithm which are executed, and the next column shows the changes which take place in the TT. For conciseness, arcless representations of all paths are used.

The algorithm begins by removing from the graph vertices a [Step ($A1$)] and h [Step ($A2$)], as well as arcs ab and gh.

The algorithm next examines path $bcd(b)$. (The parentheses around b indicate that this vertex is re-encountered, but not actually placed on the TT.) When b

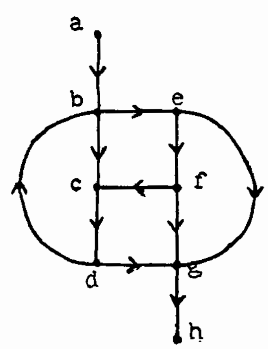

FIG. 1

[5] Two arcs are strictly parallel if they both originate on the same vertex and both terminate on the same vertex.

		TT	*P*	*Cycle-tail*		*Cycles*
1.	*A*1, *A*2, *B*	*b*				
2.	*C*1, *C*2	*bc*				
3.	*D*1, *D*2, *E*1, *E*2	*bcd*				
4.	*D*1, *D*2, *E*1, *E*2	*bcd*(*b*)				
5.	*D*1, *D*2, *E*1, *E*3	*bcd*				→*C*1. *bcdb*
6.	*F*	*bcdg*				
7.	*D*1, *D*2, *E*1, *E*2	*bcdge*				
8.	*D*1, *D*2, *E*1, *E*2	*bcdgef*				
9.	*D*1, *D*2, *E*1, *E*2	*bcdgef*(*c*)				→*C*2. *cdgefc*
10.	*D*1, *D*2, *E*1, *E*3	*bcdgef*				
11.	*F*	*bcdgef*(*g*)				→*C*3. *gefg*
12.	*D*1, *D*2, *E*1, *E*3	*b*				
13.	*F*	*b*(*e*)				
14.	*D*1, *D*2, *E*1, *E*4	*b*	*e*	*fc*	(*C*2)	
15.	(*G*1, *G*2, *G*4,	*b*	*efc*	*db*	(*C*1)	→*C*4. *befcdb*
16.	*G*6(*G*1, *G*2, *G*4, *G*7, *G*9	*b*	*efc*		(*C*2)	
17.	*G*10, *G*3	*b*	*e*	*fg*	(*C*3)	
18.	*G*10), *G*10, *G*3, *G*4	*b*	*efg*	*efc*	(*C*2)	
19.	*G*6(*G*1, *G*2, *G*4, *G*5	*b*	*efg*		(*C*3)	
20.	*G*10), *G*3	*b*				
21.	*G*10), *G*3					
22.	*F*, *C*1					

FIG. 2

is re-encountered, the cycle *bcdb* is formed (line 4), and the algorithm backs up to the last branch point, *d*. Similarly, cycles *cdgefc* and *gefg* are found on lines 10 and 12.

When vertex *e* is re-encountered (line 14), a call to *Concat* is required. (Calls to *Concat* are indicated by left parentheses and returns by right parentheses.) The tail of cycle *cdgefc* with respect to *e* is taken (line 15), and a recursive call to *Concat* is made (line 16). The tail of cycle *bcdb* is taken with respect to *c*, and, since the terminal vertex of this cycle-tail belongs to the *TT*, a new cycle, *befcdb*, is formed [Step (*G*9) of line 16].

The tail of cycle *cdgefc* is then taken with respect to *c* (line 17). However, since this cycle-tail contains no noninitial vertices, it is discarded without any further action being taken [Step (*G*3)]. At this point, all cycles containing *c* which had existed prior to the recursive call to *Concat* have been examined, and so this second-level execution of *Concat* is complete, and control is returned to the first-level execution of *Concat* (line 18). After one more second-level execution of *Concat* (lines 19–21) which does not result in finding any more cycles, the first-level execution terminates as well (line 21), and, finally the entire algorithm terminates (line 22).

4. *The Validity of the Algorithm*

In this section it will be proven that the algorithm does indeed find every cycle of a finite directed graph and lists each cycle precisely once.

4.1 SOME DEFINITIONS. A *strongly connected region* (*SCR*) of a directed graph is a subgraph such that for any ordered pair of vertices in this subgraph there exists a path which lies entirely within the subgraph and which originates on the first vertex and terminates on the second. Clearly, every cycle forms an *SCR*, and every element of an *SCR* belongs to one or more cycles. An *SCR* which is not a subgraph of any other *SCR* is a *maximal strongly connected region* (*MSCR*).

Corresponding to the definition of the tail of a path or cycle is that of the head of a path or cycle. The *head* of a simple path P with respect to some vertex v belonging to P, $Head(P, v)$, is obtained from P by taking its initial subpath terminating on v and then deleting v from this representation. The heads of a simple-cyclic path and of a cycle are defined similarly. Note that the vertex with respect to which a head or tail is taken belongs to neither the head nor the tail [i.e. $P = Head(P, v)\_v\_Tail(P, v)$].

To simplify one of the proofs somewhat it is also convenient to generalize the concept of a "tail": *the tail of a simple path P with respect to an arc a* belonging to the path is that terminal subpath of P which originates on the vertex on which a terminates.

Corresponding to the concept of the first vertex of a cycle is that of the "last arc of a cycle." The *last arc of a cycle* is that one of the cycle's arcs which is last encountered during a particular application of the algorithm. It should be observed that a cycle can only be discovered during that execution of Step ($E3$) or that call to *Concat* which follows the placing of the last arc on the TT.

The vertex on which the last arc terminates will be called the *last vertex of the cycle*. Note that the "last vertex of a cycle" is normally *not* the last of the cycle's vertices to be encountered. (Indeed, the last vertex of a cycle will frequently also be its first vertex.)

A vertex which is the first vertex of some cycle will be called simply a *first vertex*.

Consider now some first vertex v_f of a graph which is being examined. Consider the graph which is obtained from this original graph by deleting all arcs and vertices which were examined prior to encountering v_f. The derived graph must, of course, contain those cycles of which v_f was the first vertex, and so v_f must belong to an $MSCR$ of this graph. The SCR of the original graph corresponding to this $MSCR$ will be called the (strongly connected) *region spanned by v_f*, and will be denoted by $R(v_f)$.

Clearly, if a first vertex belongs to the region spanned by another first vertex, then the region spanned by the former first vertex is a subregion of that spanned by the latter. The *order of a first vertex v_f* will be defined recursively in terms of the orders of any other first vertices belonging to $R(v_f)$: if no other such first vertices exist then v_f is of order zero; otherwise, the order of v_f is one greater than the maximum of the orders of the other first vertices belonging to $R(v_f)$.

Finally, a vertex on which two or more arcs terminate will be called a *merge point*. Note that a first vertex which was not used as a "starting vertex" [i.e. not selected by Step ($C1$)] must be a merge point. Furthermore, if the last vertex of a cycle is not identical to the first vertex of the cycle, then the last vertex must also be a merge point.

4.2 THE VALIDITY PROOF. With the aid of two lemmas, it will now be possible to prove the following theorem.

THEOREM. *The algorithm of this paper will find every simple cycle of an arbitrary finite directed graph.*

LEMMA 1. *Suppose that the element on which some simple path P originates is the first element of the path to be placed on the TT, and that the element on which this path terminates is the last element of the path to be placed there. Then, at the time that this last element is added to the TT, P will be a terminal subpath of the TT.*

PROOF. Assume that there are n elements on P, and let them be numbered in the order of their appearance: $e_1, e_2, \cdots, e_n$. Also, let Q be a list of these elements in the order in which they are actually added to the TT; and let j be the largest number $\leq n$ such that, for each i between i and j, e_i is the ith element of Q. Since by hypothesis e_1 is the first element of Q, $j \geq 1$.

Assume that $j < n$, and let e_k be the element which follows e_j on Q. Clearly, e_j must be a vertex (in fact, it must be a branch point). Hence, e_{j+1} must be the arc of P which originates on e_j. e_{j+1} can be added to the TT only at a time when e_j is the last element of the TT. However, it is being assumed that, before e_{j+1} is added to the TT, e_k is added. Therefore, e_k must be removed from the TT before e_{j+1} can be added.

It can be seen that no element is ever removed from the TT until all paths which originate on it have been explored either to completion or until some vertex is encountered which has previously been on the TT. In particular, e_k is placed on the TT before any element of $Tail(P, e_k)$ is placed there. Therefore, all elements of $Tail(P, e_k)$ must be placed on the TT before e_k can be removed. In particular, e_n must be placed on the TT before e_k can be removed, and hence before e_{j+1} can be added. But this would violate the hypothesis that e_n is the last element of P to be placed on the TT. Therefore, $j = n$.

By the above argument it can be seen that, except for e_1, an element of P can be added to the TT only when the preceding element of P is the last element of the TT. Therefore, when e_n is added to the TT, the TT must terminate in $e_1e_2 \cdots e_{n-1}e_n$.

COROLLARY 1. *Consider a simple cycle whose first vertex and last vertex are identical. Such a cycle will be discovered by Step (E3) of the algorithm immediately after the last arc of the cycle has been placed on the TT.*

PROOF. Consider an arbitrary cycle $C = v_1a_1v_2a_2 \cdots v_na_nv_1$ $(n \geq 1)$, whose first vertex is v_1. Assume that v_1 is also the last vertex of C, and hence that a_n is its last arc. $Head(C, v_1) = v_1a_1 \cdots v_na_n$ is simple. Therefore, by the lemma, after a_n is added to the TT, the TT will terminate in $Head(C, v_1)$. At this time v_1 will be re-encountered, Step $(E3)$ will be executed, and the cycle $v_1\_Tail(TT, v_1)\_v_1 = Head(C, v_1)\_v_1 = C$ will be discovered.

COROLLARY 2. *Consider a simple cycle C_x, whose first vertex v_1 and last vertex v_x are not identical. Assume also that there exists another simple cycle C, such that:*

(i) $v_x \in C$,

(ii) $Tail(C, v_x) = Tail(C_x, v_x)$,

(iii) *C is discovered before the last arc a_x of C_x is encountered.*

Then C_x will be discovered by the call to Concat which will follow the placing of a_x on the TT.

PROOF. The first element of $Head(C_x, v_x)$ to be added to the TT is, of course, v_1; and the last element is a_x. Therefore, by the lemma, when a_x is added to the TT, $v_1\_Tail(TT, v_1)$ will be identical to $Head(C_x, v_x)$.

After a_x is added to the TT, v_x is re-encountered and *Concat* is called. Since C has already been discovered, $Tail(C, v_x)$ is one of the cycle-tails to be considered by *Concat*. Hence, the cycle $v_1\_Tail(TT, v_1)\_v_x\_Tail(C, v_x) = C_x$ will be discovered by *Concat*.

LEMMA 2. *Consider a simple cycle C_x whose first vertex v_1 and last vertex v_x are not identical. Assume also that v_1 is the only first vertex on $Tail(C_x, v_x)$. Then there must exist another simple cycle C such that:*

(i) $v_x \in C$,

(*ii*) $Tail(C, v_x) = Tail(C_x, v_x)$,

(*iii*) *C is discovered before the last arc* a_x *of* C_x *is encountered.*

PROOF. Let $P_x = Tail(C_x, v_x)$, and let m be the number of merge points, other than v_f, belonging to P_x. The proof will be by induction on m. Assume that $m = 0$. At some time after v_1 has been placed on the TT, v_x is first encountered and added to the TT. One of the paths emanating from v_x is the path P_x. Since the vertices of P_x all belong to C_x, none of them could have been encountered before v_f was placed on the TT. Furthermore, since, except for v_1, none of the vertices of P_x are merge points, none of these vertices could have been encountered between the placing of v_1 on the TT and the placing of v_x on the TT. Therefore, before v_x can be removed from the TT, path P_x must be explored until v_1 is re-encountered.

At this time a simple cycle, C_y, will be discovered [Step ($E3$)] such that $Tail(C_y, v_x) = P_x = Tail(C_x, v_x)$. Furthermore, the last of this cycle's arcs to be encountered belongs to P_x and hence to C_x. But this arc is distinct from a_x (the last arc of C_x). Hence, a_x cannot yet have been encountered. Therefore the cycle C_y has the properties required of C in the statement of the lemma.

Now assume that the proposition holds for $m < n$, and consider a cycle, C_x, for which $m = n$. Again, at some time v_1 will be placed on the TT, and, at some subsequent time, v_x will be added to the TT. While v_x is on the TT the path P_x must be explored until some previously examined vertex is reencountered. If this vertex is v_f then a cycle, C_y, will immediately be discovered such that $Tail(C_y, v_x) = P_x$.

On the other hand, suppose that this vertex is not v_1 but some other previously encountered vertex, v_y. Then, if $Tail(TT, v_1)$ is a tail of the TT which exists when v_y is re-encountered, $v_1\_Tail(TT, v_1)\_v_y\_Tail(P_x, v_y)$ forms a cycle. Call this cycle C_y. It can be seen that $Tail(C_y, v_x) = P_x$. It will be shown that C_y must be simple.

Clearly, $Tail(TT, v_f)$ and $Tail(P_x, v_y)$ must each be simple. Therefore, if C_y is not simple, there must exist one or more vertices, other than v_1, at which these two paths intersect. Let v_z be the first such vertex to have been placed on the TT. Then $Tail(TT, v_z)\_v_z\_Head(P_x, v_z)$ constitutes a cycle whose first vertex is v_z. If this cycle is not itself simple then it clearly must contain a simple cycle whose first vertex is also v_z. In either event, a simple cycle whose first vertex is v_z must exist, in contradiction to the hypothesis that the only first vertex on P_x is v_1. Therefore, C_y must be simple.

Clearly, the last arc of C_y to be encountered is the arc which terminates on v_y. Since v_x does not belong to $Tail(C_y, v_y)$ [$= Tail(P_x, v_y)$], $Tail(C_y, v_y)$ contains fewer merge points than does $Tail(C_x, v_y)$. Therefore, by the induction hypothesis, a cycle whose tail with respect to v_y is identical to $Tail(C_y, v_y)$ must have been discovered before the arc terminating on v_y is placed on the TT. Hence, by Corollary 2 to Lemma 1, C_y will be discovered by the call to *Concat* which occurs when v_y is re-encountered.

The preceding four paragraphs have shown that at some time prior to removing v_x from the TT a cycle C_y will be discovered such that $Tail(C_y, v_x) = P_x$. It follows from Lemma 1 that when a_x is placed on the TT, the only elements of C_x which are on the TT are those which belong to $Head(C_x, v_x)$. Therefore, v_x must be removed from the TT before a_x is placed on it. Hence C_y must be discovered before a_x is encountered, and so C_y has all the properties required of C in the statement of the lemma. Therefore, the lemma holds for all m.

PROOF. (Validity Theorem.) The validity theorem will be proven by proving a slightly stronger result: the algorithm will find each simple cycle of a finite directed graph before the first vertex of the cycle is removed from the TT.

The first vertex of a cycle is the last element of the cycle to be removed from the TT. By Corollary 1 to Lemma 1, each cycle whose first and last vertices are identical will be found before its last arc is removed from the TT, and hence before its first vertex is so removed.

Consider some simple cycle C_x whose first and last vertices are not identical, and let v_f and v_x be the first and last vertices respectively. It will be shown by induction on the order of v_f that C_x must be discovered before its first vertex is removed from the TT.

If v_x is of order 0, then it is the only first vertex on C_x, and hence on $Tail(C_x, v_x)$. Therefore, by Lemma 2 and Corollary 2 to Lemma 1, C_x will be discovered by *Concat* after the last arc of C_x is placed on the TT.

Now assume that every cycle whose first vertex is of order $<n$ is discovered before its first vertex is removed from the TT, and let C_x be a cycle whose first vertex is of order n. Let $P_x = Tail(C_x, v_x)$. Since $v_f \in P_x$, P_x contains at least one first vertex which spans a region containing v_x. Let v_1 be the first such first vertex.

If $v_1 = v_f$ then, by Lemma 2 and Corollary 2 to Lemma 1, C_x will be discovered by the call to *Concat* which follows the placing on the TT of the last arc of C_x.

Assume that $v_1 \neq v_f$. Since $v_x \in R(v_1)$, $R(v_1)$ must contain a cycle C_1 which contains v_x and which is such that $Tail(C_1, v_x) = Head(P_x, v_1)\_v_1$. After the last arc of C_x is placed on the TT, the last vertex v_x is re-encountered and a call is made to $Concat(v_x)$. At this time, all elements of P_x, except v_x, must already have been on the TT and been removed. In particular, v_1 must already have been so removed. Also, since $v_1 \in R(v_f)$, the order of v_1 is less than n. Hence, by the inductive hypothesis, all cycles of $R(v_1)$, including C_1, must already have been found. Therefore, during this call to *Concat*, a cycle-tail $= Head(P_x, v_1)\_v_1$ will be examined and a recursive call to *Concat* will be made using $v_x\_Head(P_x, v_1)\_v_1$ as its argument.

Let $P_1 = Tail(P_x, v_1)$ and let v_2 be the first of the first vertices which spans a region containing v_1. Then, by the above argument, at the time that the recursive call to *Concat* is made, a cycle C_2 whose tail with respect to v_1 is identical to $Head(P_1, v_2)\_v_2$ must already have been found. If v_2 is identical to v_f then during this recursive execution of *Concat* a cycle will be formed

$$\begin{aligned} &= v_f\_Tail(TT, v_f)\_v_x\_Head(P_x, v_1)\_v_x\_Head(P_1, v_f)\_v_f \\ &= v_f\_Tail(TT, v_f)\_v_x\_Head(P_x, v_1)\_v_1\_P_1 \\ &= v_f\_Tail(TT, v_f)\_v_x\_P_x = C_x . \end{aligned}$$

If v_2 is not identical to v_f, then, since P_x must be simple, another recursive call to *Concat* will be made; the argument of this call will be:

$$v_x\_Head(P_x, v_1)\_v_1\_Head(P_1, v_2)\_v_2 = v_x\_Head(P_x, v_2)\_v_2 .$$

Because the graph is finite, P_x can contain only a finite number of vertices, and so the depth of recursion must be finite. It can be seen that the argument of each of the recursive calls to *Concat* is of the form $v_x\_Head(P_x, v_n)\_v_n$. Furthermore, by the above argument, each of these recursive executions of *Concat* either results in another recursive call or in the discovery of a cycle of the form:

$$v_f\_Tail(TT, v_f)\_v_x\_Head(P_x, v_n)\_v_n\_Tail(P_x, v_n).$$

Since the depth of recursion is finite, a cycle of this form must eventually be discovered. But such a cycle can be seen to be equivalent to C_x. Therefore, C_x is discovered before its last arc is removed from the TT, and hence before its first vertex is so removed. Therefore the theorem holds.

4.3 A COMMENT. A problem related to that of finding the simple cycles of a directed graph is that of finding its *MSCR*'s. The above proofs suggest that the algorithm which has been presented can be modified so that it will find the *MSCR*'s of a graph instead of or in addition to the simple cycles. This is indeed the case, and a modification of the algorithm which will find both the *MSCR*'s and the simple cycles of a directed graph may be of some value. However, the problem of finding the *MSCR*'s of a graph is considerably simpler than that of finding its cycles, and simpler algorithms can be used if it is only desired to find the *MSCR*'s. (Such an algorithm is given in [7].) One might think of the problem of finding the *MSCR*'s as requiring less detailed structural information about a graph than the problem of finding the simple cycles.

4.4 SINGLE LISTING OF CYCLES. In Section 4.2 it was proven that the algorithm will find every cycle of a finite directed graph. It remains to show that each cycle is listed only once.

It has already been observed that a cycle can be discovered only during that execution of either Step ($E3$) or *Concat* which follows the placing of the last arc of the cycle on the TT. If the first and last vertices of a cycle are identical then *Concat* will never be called and the cycle will be found by Step ($E3$).

Consider a cycle C_x whose last vertex v_x is not identical to its first vertex. After the last arc of such a cycle is encountered, *Concat* will be called. If $Tail(C_x, v_x)$ is identical to the tail with respect to v_x of one or more other cycles, then C_x will be found when this execution of *Concat* concatenates one of these latter cycle-tails to the tail of the TT. However, since Steps ($G3$) and ($G4$) prevent *Concat* from fully examining more than one member of any set of identical cycle-tails, this execution of *Concat* can find C_x only once. Furthermore, should this execution of *Concat* result in any recursive calls to *Concat*, *Recur* will be set to 1 when the first such call is made; any cycles which are found between the start of the first recursive execution of *Concat* and the termination of the top-level execution of Concat will be compared [by Step ($G9$)] to all cycles which have been found since the start of the top-level execution of *Concat*. Since C_x cannot be found more than once prior to the first recursive call to *Concat*, it cannot be listed more than once during the entire top-level execution of *Concat*.

5. *An Evaluation*

The results of the Snobol3 implementation of the algorithm indicate that it is reasonably fast to apply. Table I summarizes the cost of applying this program on the IBM 7094 to some of the graphs tested. It should be borne in mind that the Snobol3 implementation is an experimental one, and that it is normally expected that a production implementation will be one or two orders of magnitude faster than a Snobol3 implementation.

The graph referred to as "A" in Table I is the graph given in Figure 1, and the graph referred to as "B" is given in Figure 3. The remaining two graphs were obtained from the literature: "C" is a graphical representation of the computer pro-

TABLE I

	A	*B*	*C*	*D*
No. of vertices	8	14	19	193
No. of arcs	11	23	29	294
No. of cycles	4	8	120	44
No. of MSCR's	1	2	1	17
No. of feedback arcs*	3*	7	1	23
No. of cycles in largest MSCR	4	7	120	5
Maximum no. of feedback arcs in an MSCR	3	6	1	2
Time to find cycles	0.38	0.55	28	6.8
Avg. time to find one cycle	0.095 secs	0.069 secs	0.234 secs	0.155 secs

* In the context of this algorithm, a "feedback arc" may be defined as an arc which originates on a vertex which is not examined until after the vertex on which it terminates has been examined. It should be noted that, using this definition, the feedback arcs of a graph are not a function of the graph but of how the paths of the graph are examined; e.g. the algorithm examines the graph of Figure 3 in such a way that it encounters three feedback arcs (*db*, *fc*, and *fg*), however, a different examination would produce only two feedback arcs: *db* and *ge*.

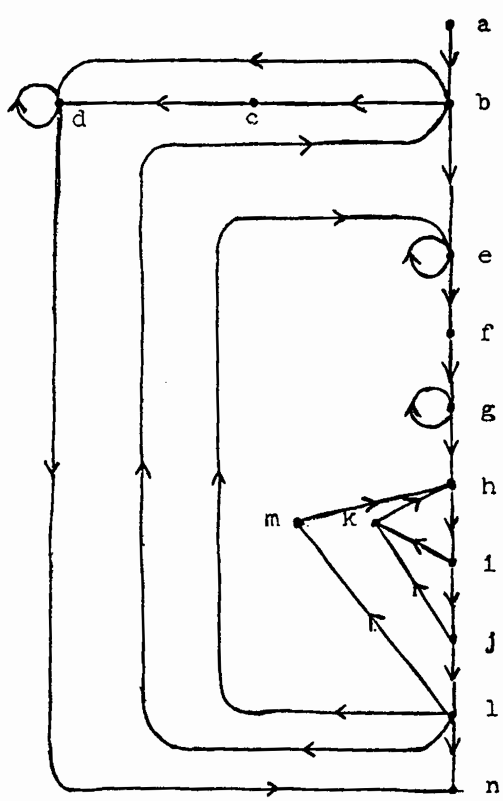

FIG. 3

gram examined in Appendix II of [1]; and "D" is the graph referred to as "$L2$" in Figure 4 of [5].

There appear to be four factors which affect the time required to examine a graph. They are the number of arcs, the number of cycles, the number of cycles per maximal strongly connected region, and the number of recursive calls to *Concat* which are required. These last two factors may be thought of as being a measure of the "complexity" of the *MSCR*'s.

Potentially the largest component of the space required by this algorithm is that required to store representations of each of the cycles which are found. In addition, representations of the graph and the TT must be stored, as well as certain information which must be associated with each vertex of the graph: the state of the vertex and a list of the cycles that have been found containing the vertex.

ACKNOWLEDGMENT. The author is grateful to A. J. Goldstein for several helpful suggestions regarding the content of this paper and for stimulating the construction of a validity proof which is substantially more illuminating than the one which had originally been devised.

REFERENCES

1. ALLEN, F. E. Program optimization. Research Report RC-1959, IBM Watson Research Center, Yorktown Heights, N.Y. (April 1966).
2. BERGE, C. *The Theory of Graphs and its Applications.* Wiley, New York, 1962.
3. BUSACKER, R. G., AND SAATY, T. L. *Finite Graphs and Networks.* McGraw-Hill, New York, 1965.
4. HAMBURGER, P. On an automatic method of symbolically analyzing times of computer programs. *Proc. 21st ACM Nat. Conf.*, ACM Pub. P-66, Thompson Book Co., Washington D.C., 1966, pp. 321–330.
5. MARTIN, D., AND ESTRIN, D. Models of computational systems—cyclic to acyclic graph transformations. *IEEE Trans. EC-16*, 1 (Feb. 1967), 70–79.
6. PROSSER, E. T. Applications of Boolean matrices to the analysis of flow diagrams *Proc. AFIPS 1959 Eastern Joint Comput. Conf.*, Spartan Books, Washington, D.C., pp 133–138.
7. RAMAMOORTHY, C. V. Analysis of graphs by connectivity considerations. *J. ACM 13* 2 (Apr. 1966), 221–222.
8. RAMAMOORTHY, C. V. The analytic design of a dynamic look ahead program and segmenting system for multiprogrammed computers. *Proc. 21st ACM Nat. Conf.*, ACM Pub P-66, Thompson Book Co., Washington, D.C., 1966, pp. 229–239.
9. RAMAMOORTHY, C. V. A structural theory of machine diagnosis. *Proc. AFIPS 1967 SJCC.* AFIPS Press, Montvale, N.J., pp. 743–756.
10. SALWICKI, A. On the application of graph theory to determine the number of multisection loops in a program. *Algorytmy 2*, 3 (1964), 73–81 (English ed.).
11. SCHURMANN, A. The application of graphs to the analysis of distribution of loops in a program. *Inform. Contr. 1* (1964), 275–282.
12. TIERNAN, J. C. An efficient search algorithm to find the elementary circuits of a graph *Comm. ACM 13*, 12 (Dec. 1970), 722–727.

RECEIVED APRIL 1969; REVISED MAY 1971