Abstract:

Let D be a directed graph, with N vertices and M edges. The best known time bound for an algorithm to enumerate the elementary cycles of D is O(N.M) per cycle. This paper presents an algorithm that requires O(N+M) time per cycle. Its space is bounded by O(N+M).

Finding the elementary cycles of a directed graph in O(N+M) per cycle

by

Jayme L. Szwarcfiter
and
Peter E. Lauer

TECHNICAL REPORT SERIES

Series Editor: Dr. B. Shaw

Number 60
May, 1974

Printed and published by the University of Newcastle upon Tyne, Computing Laboratory, Claremont Tower, Claremont Road, Newcastle upon Tyne, NE1 7RU.

bibliographical details

SZWARCFITER, Jayme Luiz

Finding the elementary cycles of a directed graph in O(N+M) per cycle. [By] Jayme L. Szwarcfiter and Peter E. Lauer.

Newcastle upon Tyne: University of Newcastle upon Tyne, Computing Laboratory, 1974.

(University of Newcastle upon Tyne, Computing Laboratory, Technical Report Series no. 60).

12"

Added entries

LAUER, Peter Ernst
UNIVERSITY OF NEWCASTLE UPON TYNE.
Computing Laboratory. Technical Report Series no. 60.

Suggested classmarks (primary classmark underlined)

Library of Congress:
Dewey (17th):
U.D.C.: 511.5
518.4

Suggested keywords

ALGORITHM
BACKTRACKING
CIRCUIT
DEPTH-FIRST
DIGRAPH
GRAPH
SEARCH

Abstract

Let D be a directed graph, with N vertices and M edges. The best known time bound for an algorithm to enumerate the elementary cycles of D is O(N.M) per cycle. This paper presents an algorithm that requires O(N+M) time per cycle. Its space is bounded by O(N+M).

About the Author

Mr. Szwarcfiter is a lecturer at Universidade Federal do Rio de Janeiro, and currently working for the Ph.D. in the Computing Laboratory, University of Newcastle upon Tyne, supported by the Conselho Nacional de Pesquisas, Brazil.

Dr. Lauer has been a lecturer in the Computing Laboratory of the University of Newcastle upon Tyne since 1972, before when he was with IBM Laboratory, Vienna.

bibliographical details

1. Introduction

Some problems, such as determining whether a graph has certain properties, or constructing a set of objects related to the graph admit of algorithmic solutions, which have a time bound linear in the size of the graph. These include the problems of finding: the strongly connected components of a directed graph (Tarjan [6]), the biconnected components of a graph (Tarjan [6]), the line graph of a graph (Roussopoulos [5]), partitions of a graph into simple paths (Hopcroft & Tarjan [2]), and others. Clearly such algorithms must be, at least proportional to the number of objects to be obtained or tested. If this number grows exponentially with the size of the graph, the algorithm will also be exponential. However, we can additionally evaluate its performance by obtaining a time bound per object generated. If this time is linear in the size of the graph, we would have achieved an optimal algorithm. The problem of finding all elementary cycles of a directed graph falls into the above category. The best known time for an algorithm solving this problem is $O(N.M)$ per cycle found, where N and M are the number of vertices and edges, respectively, of the directed graph. Such an algorithm was first obtained by Tarjan [7], but the question of the existence of a linear solution remained unsolved. The algorithm proposed in the present paper is based on Tarjan's and resolves this question in the affirmative.

A directed graph (digraph) $D(V,E)$ is a finite non-empty set V together with a binary relation E on V. The elements of V and E are the vertices and edges, respectively, of D. Given an edge $e \in E$ such that $e = (v_1, v_2)$, where $v_1, v_2 \in V$, we say that e is from vertex v_1 to vertex v_2. A sequence of vertices $v_1, \ldots, v_k$, such that $(v_i, v_{i+1}) \in E, 1 \leq i < k$, is called a path and v_1 is said to reach v_k. The path is elementary if it contains no vertex twice. A cycle is a path $v_1, \ldots, v_k$, $k>1$ with $v_1 = v_k$. A cycle $v_1, \ldots, v_{k-1}, v_k$ is elementary if $v_1, \ldots v_{k-1}$ is an elementary path. If a digraph has no cycles it is called acyclic. Two cycles are identical if they are mutually obtainable by cyclic permutations. Our problem is to present an algorithm for finding all non-identical cycles of a digraph.

Some existing cycle algorithms form the topic of Section 2. Section 3 presents our proposed algorithm which is shown to be correct in Section 4. Time and space considerations are discussed in Section 5 and further comments and conclusions form the last section.

2. Existing Methods

Tiernan [8] finds all elementary paths $v_1,\ldots,v_k$, $v_1<v_i$, $1<i\leq k$ and $1\leq k\leq N$. If $(v_k,v_1)\in E$ this cycle is recorded. Weinblatt [9] also searches for elementary paths but proposes to improve execution time by storing cycles already found and constructing new ones from these. In [7] Tarjan gives examples illustrating that both algorithms may take exponential time in the number of cycles obtained. Lauer [4] generalizes Tiernan's algorithm to different representations of digraphs, improves storage requirements and proposes alternate proofs.

Tarjan's improved algorithm [7] is based on Tiernan's depth first method. It makes use of two stacks, the point stack for storing the path currently being examined and a mark stack, as well as a boolean vector called mark vector. The mark stack is used as a stack of pointers to the mark vector. Whenever a new cycle is found all the vertices in the current point stack will eventually be unmarked when popped from this stack. If no cycle is found involving a vertex, it will be deleted from the point stack, but continue to be marked. Some of the unnecessary work done by Tiernan is avoided by the condition that a vertex that is marked and is not in the point stack when reached, will not be re-considered, at this stage. However, Tarjan followed Tiernan's principle of only searching for elementary cycles $v_j,\ldots,v_k$, with $v_j<v_i$, $1<i\leq k$, where v_j is called the start vertex. Also, whenever a vertex v is going to be unmarked because a cycle involving it was found, all the vertices that are above v in the mark stack, will also be unmarked, even if some of them are involved in no cycle.

Figures 1 and 2 present examples where unnecessary work is done by this algorithm, causing it to take quadratic time. By analysing such examples the authors realized the possibility of proposing a linear algorithm. In the acyclic digraph of figure 1,

Figure 1

the algorithm would consider vertex 1, as start vertex, obtain the path 1,...,n and would not re-explore any other vertex, at this time. But, when start vertex 2 is going to be considered, all vertices become unmarked, so path 2,...,n will be generated, although no cycles exist, and so on. n(n+1)/2 steps are required to conclude that the digraph is acyclic. In the digraph of figure 2, with one cycle, this cycle could

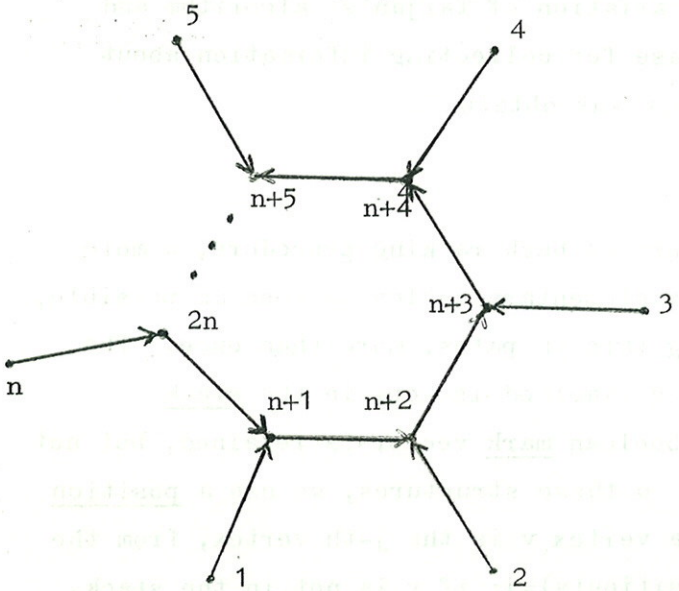

Figure 2

be detected in all computations with start vertex 1,...,n, but will not be considered since the algorithm only records elementary cycles involving the start vertex. The cycle is only recorded when n+1 is the start vertex, but paths of this cycle are again re-explored, afterwards. The algorithm requires 3n(n+1)/2 steps for completion.

Figures 1 and 2 were examples where the unnecessary work was represented by extra steps with different start vertices. Figure 3 shows unnecessary work done even within the scope of the same start vertex. Assume that edge (1,2) precedes (1,3) in the adjacency list of vertex 1, and edge (2,3) precedes (2,1) in the adjacency list of vertex 2. Then, the algorithm will generate the path 1,...,n, and nodes n,n-1,...,3 will be popped from the point stack, with the mark on. But, when edge (2,1) is explored a cycle is found, and since 3,...,n are all above 2 in the mark stack, at that moment, all 3,...,n

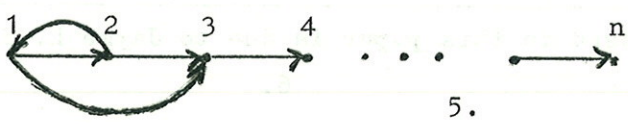

Figure 3

will be unmarked. This means that the path 1,3,4,...,n will also be generated, although 3,...,n has already been traversed and no cycles were found with them. The algorithm proposed in the present paper avoids the unnecessary work illustrated in figures 1,2 and 3.

Another method was developed later by Ehrenfeucht, Fosdick & Osterweil [1] which is a variation of Tarjan's algorithm and makes use of an additional phase for collecting information about the digraph. But no better time was obtained.

3. The proposed algorithm[1]

Our algorithm uses a recursive backtracking procedure, a more efficient system for detecting elementary cycles as soon as possible, and avoids unsuccesfully going through paths, more than once. The elementary path currently being examined is kept in the stack (Tarjan's point stack). The boolean mark vector is retained, but not the mark stack. In addition to these structures, we use a position vector and reach vector. If a vertex v is the j-th vertex, from the bottom of the stack, then position(v)=j; if v is not in the stack, then position(v)=N+1. Before a vertex v leaves the stack for the first time reach (v)=N+1. When v is popped from the stack reach (v) is set to the vertex at the bottom of the stack (start vertex). A vertex is marked when it enters the stack, and the mark is kept as long as the vertex remains in the stack. A vertex is unmarked when and only when it is popped from the stack and a new elementary cycle involving this vertex, but not necessarily the start vertex, was found. If a vertex leaves the stack and remained marked, it will never again be re-explored.

The digraph is supposed to be given as a set of adjacency lists, one for each vertex. If (v,w) is an edge of the digraph this corresponds to one node j in the adjacency list of vertex v, with vertex (j)=w and link (j) pointing to the next node of this list, if there exists one. Otherwise if j is the last node of this list then link (j)=0. For an empty adjacency list of vertex v, top(v)=0. The vertices of the digraph are assumed to be {1,...,N}.

---

[1] The algorithm proposed in this paper is due to Jayme L. Szwarcfiter.

The basic idea of the algorithm is similar to all previously described methods, namely to try to extend the current elementary path in examination. Consider the case where vertex 1 is the start vertex, vertex v is at the top of the stack and edge (v,w) is reached:

(i) If w is unmarked then necessarily, w is not in the stack, the elementary path will be extended with w, and an edge from w will be examined.

(ii) If w is marked and not in the stack then, necessarily, there can be no new cycles with w, and w will not be re-explored.

(iii) If w is marked and in the stack then a cycle was found, and it can be recorded at once. The algorithms of Tiernan, Lauer Tarjan and Ehrenfeucht & al., all disregard these cycles, if w is not the start vertex. The problem that arises when considering such cycles is that a mechanism for detecting duplicate cycles must be set up. The nature of this mechanism follows from the observation that a cycle is a new cycle, if and only if at least one of its vertices had never been deleted from the stack. The fact that it has not been deleted before is indicated by setting a variable q corresponding to the vertex v, such that position(w)<q.

In cases (ii) and (iii), when w is marked, the elementary path is not extended. If a certain elementary path can not be extended any more, the algorithm backtracks to the previous vertex in the stack, and so on. When the start vertex is deleted from the stack, only vertices v, with reach(v)=N+1 will be considered as candidates for being new start vertices, and in the subsequent steps, only vertices with reach(v)≥ current start vertex, will be considered for examination. This follows from the fact that if a vertex v was first reached when the start vertex was j, then all the cycles involving v were enumerated before j left the stack.

The program below uses the simple variable t as a pointer to the top of the stack. The combined action of f and g assures that if a new cycle is found involving the k top vertices of the stack, then all k vertices will be unmarked upon leaving the stack. If no new cycle was detected

they remain marked on deletion. The global variable j represents within the procedure CYCLE, the current start vertex. N and M are, respectively, the number of vertices and edges of the considered digraph.

```
begin comment program for finding the cycles of a digraph;
      procedure CYCLE (integer value v,q; integer result f);
      begin integer g,p,w;
            f := N+1;
            t := t+1;
            stack(t) := v;
            position(v) := t;
            mark(v) := true;
            if reach(v) = N+1 then q := N+1
            else if q = N+1 then q := t;
            p := top(v);
            while p≠0 do
            begin w := vertex(p);
                  if reach(w)≥j then
                  begin if ¬mark(w) then
                        begin CYCLE (w,q,g);
                              if g<f then f := g;
                        end
                        else if position(w)<q then
                              begin output cycle v to w, from the stack;
                                    f := position(w);
                              end;
                  end;
                  p := link(p);
            end;
            if position(v)≥f then mark(v) := false;
            t := t-1;
            reach(v) := j;
            position(v) := N+1;
      end CYCLE;
      integer array reach, top, stack, position(1::N);
      logical array  mark(1::N);
      integer array vertex, link(1::M);
      integer j,t, dummy;
      t := 0;
      for j := 1 until N do
      begin top(j) := 0;
            mark(j) := false;
            position(j) := reach(j) := N+1;
      end;
      read the digraph and construct the adjacency lists;
      for j := 1 until N do if reach(j) = N+1 then CYCLE (j, dummy, dummy);
end.
```

4. Correctness

Let D be a digraph with N vertices, input to the program:

Lemma 1: If $v_1,\ldots,v_k,v_1$ is an elementary cycle of D, then there exists a configuration of the stack, such that a cyclic permutation of $v_1,\ldots,v_k$ appears as the k top positions of it.

Proof: Induction on K. If K=1 the lemma holds because every vertex enters the stack at least once, since a simple inspection of the program shows that every vertex appears as the first parameter of CYCLE, at least once. Assume, without loss of generality, that v_1 was the first vertex, among $v_1,\ldots,v_k$, to enter the stack, and by the induction hypothesis $v_1,\ldots,v_{k-1}$ was generated for the first time and occupies the k-1 top positions of the stack, with v_{k-1} at the top. Therefore, v_1 has never left the stack, and reach(v_1) = N+1. Vertex v_{k-1} is now being examined and assume edge (v_{k-1},v_k) is reached. Suppose v_k is marked: If v_k is in the stack, it is underneath v_1, which contradicts the fact that v_1 was the first vertex of this cycle to be inserted in the stack. If v_k is not in the stack, then v_k had entered it, after v_1, and left it, with no new cycles found, which contradicts the existence of the edge (v_k,v_1) and the fact that reach(v_1) = N+1. Thus, v_k is unmarked and will be placed on the top of v_{k-1}, in the stack.

Lemma 2: Each elementary cycle of D is listed at least once.

Proof: Let $v_1,\ldots,v_k,v_1$ be an elementary cycle of D. By lemma 1, a cyclic permutation of $v_1,\ldots,v_k$ will appear on the top of the stack, at some time. Then, the cycle $v_1,\ldots,v_k,v_1$ will be generated. Consider, now, the first instance of the generation of this cycle, with the k top positions of the stack being $v_1,\ldots,v_k$, with v_k at the top. Therefore, v_1 did not leave the stack before, reach(v_1) = N+1 and position(v_1)<q (corresponding to v_k). Hence, $v_1,\ldots,v_k,v_1$ will be listed.

Lemma 3: Each elementary cycle of D is listed at most once.

Proof: Let $v_1,\ldots,v_k,v_1$ be an elementary cycle of D, such that for $i\leq j\leq k$, v_j has sometime been deleted from the stack. Assume $v_1,\ldots,v_k$ are the top k positions of the stack, with v_k at the top. Since reach(v_j)$\neq$ N+1, we have position(v_j)$\geq$q (corresponding to v_j), for $1\leq j\leq k$. Then, $v_1,\ldots,v_k,v_1$ will not be listed.

Theorem 1: Each elementary cycle of D will be listed exactly once.

Proof: Lemmas 2 and 3.

5. Performance

Theorem 2: Let D be a directed graph, with N vertices, M edges and C elementary cycles, input to the program. Then the program requires $O((C+1)(N+M))$ time and $O(N+M)$ space to enumerate C elementary cycles.

Proof: The space bound follows from the fact that the representation of the directed graph, by adjacency lists, requires $O(N+M)$ cells, and the data structures of the program require $O(N)$.

For the time bound, we recall that a vertex can only be explored if it is not marked when reached, and once marked, it can only be unmarked if a new cycle is found, with this vertex. The exploration of a vertex v_j means a call of the procedure CYCLE, with $v=v_j$. If a vertex is explored, all edges starting from it are also explored. Now, assume the algorithm is in a state in which a new cycle has just been found, and let us compute the maximal work necessary to find the next new cycle. During this interval of time, vertices that were in the stack may become unmarked, and a vertex, if it enters the stack, will be marked, and remain marked, at least, during the entire interval of time considered. This prevents the vertex from being re-explored, during this period. So, a vertex can be explored at most once, between new elementary cycles. In the worst case, we have all N vertices explored, once each, and all edges, starting from each of the vertices, also explored once each. Thus, the bound is $O(N+M)$ per elementary cycle. The program would require $O(N+M)$ time to process an acyclic digraph, since no vertices can be unmarked in this case. Thus, the bound for C elementary cycles is $O((C+1)(N+M))$.

6. Conclusions and Comments

An algorithm for obtaining the cycles of a directed graph was presented. The program of section 3 can still be accelerated, if we delete nodes j from the adjacency list of vertex v, when vertex(j) was found to be marked and not in the stack, while exploring the edges of v (after f := position(w) end, include: if position(w) = N+1 then delete node p;). Also, storage can be saved, since it is possible to merge the position, reach end mark vectors, into one single vector, using positive and negative integers.

The algorithm possesses the following features:

(i) The algorithm is bounded by O(N+M) time per cycle enumerated and, clearly, no sequential algorithm can present a smaller bound.

(ii) Its data structures require O(N) cells of storage.

(iii) It operates with the digraph represented as adjacency lists, which corresponds to O(N+M) cells.

(iv) No additional pre-processing is required.

(v) A vertex that is not part of any elementary cycle is explored exactly once.

(vi) After all elementary cycles involving a given vertex have been enumerated, this vertex is explored at most once.

If the deletion of nodes of the adjacency lists, above mentioned, is incorporated, the following are also true:

(vii) Edges that do not lead to elementary cycles are explored exactly once.

(viii) After all elementary cycles involving a given edge have been enumerated, this edge is explored at most once.

The algorithm was implemented using ALGOLW and an IBM 360/67. The 125,664 cycles of the complete graph with 9 vertices and no self-loops were counted in 56.9 seconds. A non-recursive version of the program, following Knuth's techniques [3] was also implemented, and the running time for this same digraph was 49.4 seconds. These times correspond to the running of the entire programs, including input, but with the output of the cycles substituted by a counter. Tarjan's figure, for this same digraph, using the same language and a 360/65 machine was 101.2 seconds, according to [7]. Finally, we mention that when translating to the non-recursive version, observe that no stack is needed for the parameter v, since it would be redundant with the information in the stack vector. Also, no stack is required for w, since it can be transformed into a global variable. Furthermore, it is possible to alter the program, so that the stacking of parameter q would also be avoided.

REFERENCES

1. Ehrenfeucht, A., Fosdick, L.D., Osterweil, L.J.: An algorithm for finding the Elementary Circuits of a Directed Graph. Technical Report #CU-CS-024-73, Univ. of Colorado, 1973.

2. Hopcroft, J., Tarjan, R.: Efficient algorithms for graph manipulation. Comm. of the ACM, 16, 372-378 (1973).

3. Knuth, D.E.: Structured programming with go to statements (manuscript).

4. Lauer, P.E.: The perils of indirect Proof or another efficient search algorithm to find the elementary circuits of directed graphs. Technical Report 42, Comp. Lab., Univ. of Newcastle upon Tyne, 1973.

5. Roussopoulos, N.D.: A $\max\{m,n\}$ algorithm for determining the graph H from its line graph G. Inf. Proc. Letters 2 108-112 (1973).

6. Tarjan, R.: Depth-first search and linear graph algorithms. SIAM J. Comput., 2, 146-160 (1972).

7. Tarjan, R.: Enumeration of the elementary circuits of a directed graph. SIAM J. Comput., 3, 211-216 (1973).

8. Tiernan, J.C.: An efficient search algorithm to find the elementary circuits of a graph. Comm. of the ACM, 13, 722-726 (1970).

9. Weinblatt, H.: A new search algorithm for finding the simple cycles of a finite directed graph. J. of the ACM, 19, 43-56 (1972)