A SEARCH ALGORITHM FOR FINDING THE SIMPLE CYCLES OF A DIRECTED GRAPH

BY

LeRoy Johnson

TR74-001, February 1974

A SEARCH ALGORITHM FOR FINDING THE SIMPLE CYCLES OF A DIRECTED GRAPH

LeRoy Johnson

University of New Brunswick

Fredericton, Canada

TR74-001, February 1974

ABSTRACT

Two efficient algorithms, that admit each simple cycle once in searching the arcs of a digraph, are presented and a proof of the algorithms provided. The first algorithm is nonadaptive while the second is an adaptive one. Since they extract cycles directly, they require less storage than Weinblatt. Since the method of search of the first is more efficient than Tiernan, it is faster than Tiernan. An example used by Tiernan is given for comparison. The speed is discussed in relation to number of vertices, arcs, and simple cycles. The storage requirement is $O(n)$. The adaptive version has a theoretical speed bound by $O(c.m.n)$, where c is the number of cycles, m the number of arcs, and n the number of vertices. Both algorithms are bounded by $O(N_c.m)$ on the complete graph where N_c is the number of cycles of the complete graph.

1. INTRODUCTION

In recent years many papers have been written that consider algorithms for finding cycles of graphs and digraphs or that require such algorithms [2,3,4,5,6,7,8,10].

For graphs one approach has been to generate a fundamental set of circuits and, since it is known for undirected graphs that all linear combinations of a fundamental set produce all the circuits, then to generate the set of simple circuits from this result. Unfortunately, this result does not apply to digraphs and so other methods have been proposed.

Recently, notable algorithms have been presented by Tiernan [8] and Weinblatt [9] which apply search techniques to find the simple cycles of digraphs. Tiernan claims that he only considers each cycle once. This is correct in that he only finds or reports a cycle once; however, in his algorithm the same cycle may be found implicitly many times. The author's algorithm also reports a cycle only once and may find a cycle a number of times but not so many as Tiernan.

Weinblatt examines each arc only once but does not find all cycles directly. The remaining cycles are found by combining parts of previously discovered cycles with a part of the current path being searched. The time saved by efficient search (better than ours) is lost in searching the constructed cycles and paths which must be saved in order to generate all the simple cycles; this results in increased storage requirements.

2. BACKGROUND

For our purpose a digraph $D = \langle V,A \rangle$ is composed of two sets, a set of vertices V and a set of arcs A where $A \subseteq V \times V$.

A path from vertex v_1 to v_n is an ordered sequence of vertices $(v_1,v_2,\ldots,v_n)$ such that for v_i and v_{i+1} in the sequence $\langle v_i,v_{i+1} \rangle \epsilon A$. A path whose first and last vertices coincide is called a cycle and any path which has no repeated vertex is said to be simple. A cycle is simple if no vertex except the last is repeated. Some authors give the term "elementary circuit" for our simple cycle, however, we prefer the prefix simple to that of elementary since it is obvious from elementary considerations that it is more simple.

Cycles which describe the same set of arcs will not be considered to be distinct, we can of course uniquely represent simple cycles by assuming they begin on the lowest ordered vertex, assuming some ordering on V. We shall say that a cycle $(v_1,v_2,\ldots,v_n,v_1)$ begins on v_1.

3. THE ALGORITHM

We find it convenient and we believe enlightening, to provide two statements of the algorithm; one a formal English version which usually achieves clarity at the expense of a certain amount of ambiguity. The second version is an illustration of how the algorithm might be specified unambiguously to a computer using APL.[1]

---

[1] A Fortran version is available from the author.

In essence the algorithm begins at some distinguished vertex called the KEY and begins to search out a path originating on KEY while it remembers the path in a pushdown stack. We store the path as a sequence of vertices since for our definition of a digraph this is sufficient to describe a path; however, we could use arcs, or even arcs and vertices to describe the path.

Before adding a vertex to the stack we check that it is not in the stack so that the path we search will be simple.

If the vertex is already in the stack then we have found a cycle. A simple test on information generated by the algorithm determines if this is our first discovery of this cycle. A cycle is only reported when it is first discovered. Suppose all paths originating on KEY have been searched, then if there is a vertex that has not yet been examined it becomes a new KEY and the procedure is repeated until all vertices have been examined.

The digraph of Figure 1 was used as an example by Tiernan and we also use it for our example to facilitate a comparison with his algorithm. Figure 2 illustrates the condition of the stack VS at each step and square brackets indicate a cycle was found and that the vertex is not entered on the stack. With reference to Tiernan's example, our stack is evaluated 13 times (using his method of recording) as compared to 29 times by Tiernan's method.

ALGORITHM: DCYCLE

Comment: Let D = <V,A>. Let VS and OS be pushdown stacks and V,A,A' sets.

1: Initialize

VS ← ∅; OS ← ∅; A'← ∅

KEY ← v, for some v ∈ V

2: Stack

VS ← v,VS

3: Advance

3.1 Find arc <v,w>∈A, set

A ← A - {<v,w>}

A'← A' ∪ {<v,w>}

If there does not exist such an arc, go to Step 5:Retreat.

3.2 If w ∉ VS, v ← w and go to Step 2:Stack.

3.3 If w ∈ VS and w ∉ OS, go to Step 4:Report Cycle.

3.4 Since w ∈ VS and w ∈ OS, we cannot report this cycle; go to Step 3:Advance.

4: Report Cycle

4.1 CYCLE ← Copy head of VS down to w.

4.2 Output CYCLE.

4.3 Go to Step 3:Advance.

5: Retreat

5.1 Delete v from top of VS and place on OS.

5.2 Replace arcs that originate on v;

$A \leftarrow A \cup \{<v,x>\}$

$A' \leftarrow A' - \{<v,x>\}$ for all such x.

5.3 If $VS \neq \emptyset$, then set v to the top of VS and go to Step 3:Advance.

6: Obtain Next KEY

6.1 Select vertex $v \in V$ such that $v \notin OS$, if there is none, Stop.

6.2 $KEY \leftarrow v$

6.3 Go to Step 2:Stack.

⊕

An APL Implementation

```
     ∇ DCYCLE G
[1]    N←1↑⍴G
[2]    TRY←N⍴1
[3]    VS←0
[4]    KEY←1
[5]  START:PN←KEY
[6]    PS←N⍴KEY-1
[7]  STACK:VS←PN,VS
[8]  GO:PPN←PN
[9]  EXTEND:PN←PS[PN]+(PS[PN]↓,G[PN;])⍳1
[10]   →(PN>N)/RETREAT
[11]   PS[PPN]←PN
[12]   →(~PN∈VS)/STACK
[13]   →(TRY[PN]=0)/RETRY
[14]   ⍞  REPORT CYCLE
[15]   (VS⍳PN)↑VS
[16] RETRY:PN←PPN
[17]   →EXTEND
[18] RETREAT:VS←1↓VS
[19]   PS[PPN]←KEY-1
[20] R1:TRY[PPN]←0
[21]   PN←1↑VS
[22]   →(PN≠0)/GO
[23]   →(N>KEY←TRY⍳1)/START
     ∇
```

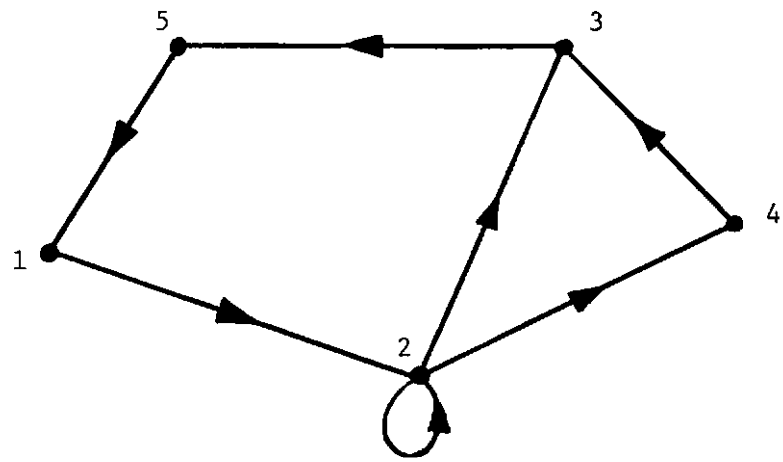

Fig. 1 A geometric representation of a digraph

VS	ACTION
	KEY ← 1
1	
1 2	
1 2 [2][1]	Report Cycle (2,2)
1 2 3	
1 2 3 5	
1 2 3 5 [1]	Report Cycle (1,2,3,5,1)
1 2 3	
1 2	
1 2 4	
1 2 4 3	
1 2 4 3 5	
1 2 4 3 5 [1]	Report Cycle (1,2,4,3,5,1)
1 2 4 3	
1 2 4	
1 2	
1	
0	

1 Note [v] is not added to stack.

Fig 2 Condition of Stack when searching digraph of Fig. 1 from Vertex 1.

4. PROOF OF THE ALGORITHM

An algorithm may be defined as a procedure that halts, so we must show that our algorithm does indeed execute the procedure claimed and requires but a finite number of steps to do so. Since the algorithm is known to sometimes examine a cycle more than once, we must insure that it can never enter a loop with no exit.

In the following we assume that D is connected; otherwise we merely apply the algorithm to each component if D is not connected.

Lemma 1 If D is finite, then the procedure of the algorithm halts after a finite number of steps.

Proof: Consider a vertex v on the top of the stack; by Step 3, v can only remain on the top of the stack for a finite number of operations. Suppose v is not removed, then some v'is added. Since a vertex can only occur on the stack once and there are a finite number of vertices, there must exist a vertex u that is the last vertex that can be placed on the stack. Now by Step 3, u must be removed.

Suppose now that every vertex v' that can follow v on the stack has been removed, then v is again on the top of the stack and by Step 3 no vertex can be placed on the stack so that v must then be removed.

Thus the first vertex on the stack, KEY, must be removed after a finite number of steps and Step 6 entered. Since a new key

cannot be a vertex that has been on the stack, the procedure must halt after all vertices have been removed from the stack.

⊕

Lemma 2 Every vertex is placed on the stack.

Proof: Assume that the Algorithm has stopped. Suppose u is not placed on the stack, then since it has not been removed, $u \notin OS$. However, then by Step 6 (a) the Algorithm has not stopped. Contradiction.

⊕

Lemma 3 Every cycle found that begins on v is reported before v has been removed from the stack.

Proof: Assume that v is in the stack VS. Since every path from v is examined before v is removed from the stack every path that ends on v will be found. Since v has not been removed from the stack $v \notin OS$ and the cycle that begins on v is reported.

⊕

Lemma 4 The Algorithm reports a cycle exactly once.

Proof: If v has not been removed, then no cycle is repeated that begins on v. Suppose v has been returned to the stack and a cycle that begins on v is found a second time, it cannot be reported since $v \in OS$. Suppose a cycle beginning on v that has been reported is now found and begins on u. Since v was removed, then u must have been removed from the stack and $u \in OS$ which contradicts the assumption that the cycle is reported.

⊕

Theorem 1 The Algorithm finds every simple cycle.

Proof: Assume false. Then there exists a simple cycle $c = (v_1,\ldots, v_n,v_1)$ that has not been found and thus neither $(v_1,\ldots,v_n)$ nor any rotation has occurred on the top of the stack. Since each vertex is placed on the stack at least once, some vertex of x must have been the first of c placed on the stack, say v_k. Then for some extension of v_k, v_{k+1} will be found and since $v_{k+1} \notin$ VS then it can be replaced on the stack. This can be repeated for each extension of v_i to v_{i+1} of c, until $v_{i+1} = v_k$ but then by construction the top of the stack is the cycle c and it will be reported.

⊕

Theorem 2 The Algorithm finds the set of simple cycles of a finite directed graph.

Proof: From Lemma 1 the procedure halts. From Lemma 4 and Theorem 1 it finds the set of simple cycles.

⊕

5. EVALUATION

The evaluation of an algorithm is generally a non-trivial problem and comparisons of algorithms even more so. Three important attributes of an algorithm are simplicity of the algorithm, storage requirements, and execution speed. The fact that these three attributes, for example, generally interact and can require trade-offs, further complicates matters. One would like a simple algorithm not only for ease of understanding but because simplicity often implies other desirable properties. As an example, when the algorithm is executed by an interpreter, simplicity can improve execution speed. It is difficult to judge the complexity of representation but we feel that our algorithms are simple and fast and certainly less complex than those of Tiernan or Weinblatt.

We will not perform an exhaustive evaluation of the algorithms but do indicate briefly their storage and speed properties.

5.1 Storage

The storage requirement often depends upon the implementation of the algorithm so we will evaluate the storage for the APL version. We assume that all words are stored in fixed length cells of unit length. First a push down stack VS is required which can grow to length n. Two vectors TRY and PS of length n are required. So the storage is $3n + c$ where c is a constant. In a Fortran version an additional vector was used to increase execution speed. Storage then is bounded by $O(n)$.

This is considerably better than Weinblatt. Although it is somewhat better than that indicated by Tiernan's description of his algorithm of $n^2 + n + c$, the author has programmed Tiernan's algorithm

using 2n + c storage and so our algorithm is comparable to Tiernan in this respect. The small difference is due to a speed versus storage trade off.

5.2 Speed

A usual method of evaluating the speed of a graph algorithm is to relate processing time to the number of vertices n. Since an algorithm determines some property of the graph,we may consider the speed in relation to the amount of property that a particular graph has. In particular, for this algorithm the complexity of the graph is determined by the number of cycles and the length of these cycles; n is not a good estimate of this.

Ideally we should generate graphs whose cycles increase in quantity and average length and plot the processing time. One may also determine a theoretical estimate mathematically. For practical reasons we have not done this in general. Another possibility is to consider the worst case and this is to consider for each n the complete digraph.

Certainly the complete graph has the most cycles and for increasing n the average cycle length increases. However, examination of the number of cycles of a complete digraph indicates that for practical purposes it is unlikely that high density digraphs will be frequently encountered. A more useful evaluation of the algorithm would result from a consideration of low density digraphs. Alas, it is difficult to randomly generate such digraphs with desirable properties. Also, since loops are simple to find, it is preferable to consider digraphs without loops.

Due to the symmetry of the complete digraph its cycles can be calculated directly. In order to count the cycles we note that each permutation of every arbitrary subset of vertices represents a cycle. These permutations do not all represent distinct cycles,for every rotation of a permutation gives the same cycle.

There are then $\frac{n!}{m((n-m)!)}$ distinct cycles of length m and thus the total number of cycles of K_n (omitting loops) is

$$N_c = n! \sum_{m=2}^{n} \frac{1}{m((n-m)!)}$$

and the average cycle length is

$$\bar{c}_\ell = \frac{\sum_{m=2}^{n} \frac{1}{((n-m)!)}}{\sum_{m=2}^{n} \frac{1}{m((n-m)!)}}$$

From the formula for N_c, it is not easy to comprehend the size of the number involved, so in Appendix I we have derived some simple bounds.

$$2((n-1)!) < N_c \leq \frac{5}{6} n! \quad n=3,4...$$

The complete digraph then gives an upper bound on the execution speed of any digraph of n vertices or any digraph of m arcs. For the execution time on a digraph must be less than that on the minimal complete graph that contains its number of arcs.

The problem with this bound is that, for low density digraphs with a good distribution of the arcs, the bound is unrealistically high as may be seen in Figure 4. Nonetheless, we indicate results on complete digraphs in Figure 4 for a Fortran version of DCYCLE run on an IBM 370 model 158 computer. It should be noted that these times include the reporting of the arcs of the cycles found. This is done by transferring the cycle from the stack to a dummy vector.

N_V	N_E	N_C	$\bar{c}_\ell$
1	0	0	0
2	1	1	2
3	6	5	2.4
4	12	20	3
5	20	84	3.8
6	30	409	4.76
7	42	2,365	5.78
8	56	16,064	6.82
9	72	125,664	7.84
10	90	1,112,073	8.87

Fig. 3

N_V	$\bar{t}$	$\bar{t}/N_c$	$\bar{t}$/cycle-arc
	sec	msec/cycle	msec
5	.011	.131	.034
6	.069	.169	.035
7	.49	.207	.036
8	4.03	.251	.037
9	36.84	.293	.037

Fig. 4 Execution time of DCYCLE on K_n: Fortran G Compiler

We can experimentally evaluate the efficiency of the algorithm on the complete digraph by the following simple strategy. First, find the execution time to discover the cycles (that is, do not report them); second, find the execution time to discover and report the cycles. Reporting a cycle consists of recording it in a vector. The difference then is the time required to report the cycles and it is clearly an unattainable lower bound on any algorithm.

For K_7 and K_8 the time to discover and report cycles is approximately three times that required just to discover cycles. One would suspect that any algorithm must require twice the time required merely to report a cycle in order to find and report, since reporting is so simple an operation. As the algorithm appears to require about three times this reporting time, it would appear that for digraphs similar to the complete digraph the algorithm is extremely efficient.

Define the search between the discovery of cycles as a cycle subsearch.

Theorem 3 The speed of DCYCLE on a complete digraph K_n is of order $O(N_c \cdot m)$.

Proof: Let D be a complete digraph. The algorithm can retreat on at most n vertices before advancing or halting. Since D is complete, there exists a path from every vertex to KEY, that does not intersect VS, therefore, once the algorithm advances it must find a new cycle before it retreats. Each cycle subsearch is then bounded by O(m) and there are N_c cycles so that the speed is of order $O(N_c \cdot m)$ on K_n.

⊕

5.3 Improvements

There are a number of possibilities for improving this algorithm; however, since the alterations are essentially extensions of the algorithm that reduce the number of times certain arcs are reconsidered, simplicity deteriorates: as the potential saving requires increased testing and in some cases additional storage, it is not yet clear what degree of improvement if any results. Most certainly performance will be degraded on the complete digraph, but then this is a trivial case, as we can calculate the results.

First of all there is no need to re-enter a strong component once it has been examined; thus, this could be forbidden by the algorithm. In certain cases this could be expensive in time and, of course, a complete disadvantage for a strongly connected digraph, especially the complete digraph. This can be done by identifying the cutvertices of the strong components and then marking as forbidden those non-cutvertices that belong to a strong component, that has been removed from the stack.

If one considers that it is parallel paths that lead to much of the reconsideration of certain arcs,one could consider techniques similar to Weinblatt's algorithm and retain certain cycles so that the information may be extracted from previous cycles.

It would be expected that the speeds of such modified algorithms may be strongly dependent on the properties of the graphs being examined. So we would like first to have some knowledge of the general properties of graphs to which the algorithm is most likely to be applied.

Although the complete digraph is the worst case for the number of cycles, it is actually not so when we consider the efficiency of finding the cycles in the digraph, since every extension following a retreat will lead to at least one new cycle. If every arc advance is in a cycle, then the algorithm is very efficient.

Consider a maximum acyclic digraph A. Since A is acyclic, then the vertices can be linearly ordered so that there are no feedback paths. Now since A has a maximal number of arcs, then there must exist arcs from each vertex to every higher ordered vertex. Thus every ordered subset of this ordered set of vertices is a simple path. Now for a given simple path, v_i is in or is not in the path; thus the number of simple paths is 2^n.

Our algorithm as stated would examine each such simple path containing KEY and so 2^{n-1} paths would be examined. This would indicate that the algorithm as stated is not polynomial, with respect to the number of cycles, in the worst case. However, if we label the vertices which we know cannot lie in a new cycle, then the algorithm would add a vertex of A to the stack at most once. Since each vertex is added to the stack at most once, consequently each edge is examined at most once; the search of A in this case, an acyclic digraph, would be O(m) and the search a linear function of the input.

It appears that the most efficient implementation of this result is to first apply an algorithm to identify the strong components of the digraph and then to apply DCYCLE to each strong component found. However, this is another problem: so in the next section we show how this result may be partially incorporated into our algorithm.

6. AN ADAPTIVE ALGORITHM

The example of the maximum acyclic digraph A in Section 5 showed that DCYCLE does not have a polynomial speed bound. We present here an algorithm with such a bound. We can modify DCYCLE by a simple strategy so that it becomes an adaptive algorithm: once an arc is examined it becomes forbidden until such time as it may occur in a new cycle by labeling it with a 1; this can only occur after a cycle is reported. The labels are cleared by setting them to 0. The increased overhead is in many cases justified by reduced search.

Theorem 4 ACD reports every simple cycle.

Proof: Suppose that ACD does not report a cycle α. Some vertex x of α must have been the first to be placed on the stack. If any vertex y of α is placed on the stack, there exists a path from y to x and thus a cycle will be reported before y is removed from the stack. This cycle has its LABELs cleared, therefore, when y is removed all vertices placed on the stack after y will have their LABELs cleared. Because of this, no vertex of α not on the stack can be forbidden until x has been removed. Therefore, by Theorem 1,α must be reported. ⊕

Theorem 5 ACD is an algorithm that reports the set of simple cycles of a finite digraph.

Proof: Lemma 1 still applies so the algorithm halts. By Theorem 4 every cycle is reported and by Lemma 4 at most once.

⊕

ALGORITHM: ACD

Comment: Let D = <V,A>. Let VS be a pushdown stack; V,A,A' sets; and TRY and LABEL functions on V.

1: Initialize

VS ← ∅; A' ← ∅

TRY(x) ← 0, LABEL(x) ← 0, for all x∈V

KEY ← v, for some v∈V

2: Stack

VS ← v,VS

LABEL(v) ← 1

3: Advance

3.1 Find arc <v,w>∈ A such that LABEL(w) = 0, set

A ← A - {<v,w>}

A'← A' ∪ {<v,w>}

If there does not exist such an arc, go to Step 5:Retreat.

3.2 If w ∉ VS, v ← w and go to Step 2:Stack.

3.3 If w ∈ VS and TRY(w) = 0 go to Step 4:Report Cycle.

3.4 Do not report cycle; go to 3.1.

4: Report Cycle

4.1 CYCLE ← Copy head of VS down to w and set LABEL to 0 for each vertex.

4.2 Output CYCLE.

4.3 Go to Step 3:Advance.

5: Retreat

5.1 Delete v from top of VS and set TRY (v) ← 1.

5.2 If LABEL (v) = 0, then clear all LABELs to 0, that were set to 1 after v was placed on VS.

5.3 Replace arcs that originate on v;

$A \leftarrow A \cup \{\langle v,x\rangle\}$

$A' \leftarrow A' - \{\langle v,x\rangle\}$, for all such x.

5.4 If VS $\neq \emptyset$, then set v to the top of VS and go to Step 3:Advance.

6: Obtain Next KEY

6.1 Select vertex $v \in V$ such that TRY(v) = 0, if there is none, Stop.

6.2 KEY ← v

6.3 Go to Step 2:Stack.

⊕

Theorem 6 The speed of ACD is bounded by $O(c \cdot m \cdot n)$, if D is not acyclic.

Proof: No arc is searched more than once unless a vertex v with LABEL(v)=0 is removed from the stack VS. After a cycle is reported, there are at most n such vertices in VS and this number cannot be increased unless a cycle is reported. Thus each vertex can be cleared at most n times in a cycle subsearch and consequently an arc is searched at most n times between reported cycles. The bound per cycle then is $O(m \cdot n)$. Therefore, ACD is bounded by $O(c \cdot m \cdot n)$.

⊕

Both algorithms will perform better if applied to strong components only. This suggests that any implementation should first preprocess the digraph to obtain the strong components.

Experimentation on random digraphs of up to 30 vertices indicates that neither algorithm is superior. Both experimentation and analysis indicate that any speed gain in the restriction to strong components should favour DCYCLE. However, because of Theorem 6 it is preferable to use ACD since one is protected against pathological cases and the cost is not high. No such pathological case was observed. This is probably due to the fact that in a strong component a certain complexity is required for a pathological case, but then the condition of Theorem 3 applies and DCYCLE becomes more efficient. The cycles of K_9 were found by ACD in 42 seconds or compared to 36 seconds by DCYCLE.

7. CONCLUSION

An efficient search algorithm for finding the set of simple cycles of a digraph has been presented, and its relation to the algorithms of Tiernan & Weinblatt noted. Our algorithm contradicts Tiernan's claim to be the theoretically most efficient search algorithm. Certainly any optimal algorithm for this problem must only accept a cycle once, but it is also important to minimize the search of the arcs and repeated examination of cycles.

The problem of an optimal algorithm is difficult not only to prove that you have one, but more so to decide just what is meant by optimal. Since Weinblatt searches the arcs exactly once, we might say that Weinblatt at least performs the optimal search of the digraph. However, since he must search his list of cycles and paths, it is mere semantics to say that he examines each arc but once. It is not difficult to see that we also can reduce the search in our algorithm by maintaining lists and this, perhaps, may improve the present algorithm, although we have not yet fully investigated the implications; it is, however, certain that the result will be more complex. Obtaining an algorithm that is optimal over all digraphs appears to be a difficult problem, and it is probable that one does not exist unless preprocessing is assumed. It is, perhaps, more fruitful to define optimality with respect to strong components.

The algorithm is practical in that the storage requirement is modest and the time per cycle found is reasonable, and indeed so is Tiernan's, contrary to his pessimistic conclusion for large digraphs (except for pathological instances such as our example of a maximal

acyclic digraph). In part, the practicality of these algorithms lies in the practicality of the problem; that is, one cannot find, say, 20,000 cycles in a digraph without incurring reasonable computing time in respect to the magnitude of the problem.

As a point of interest this algorithm was developed as a result of reading Weinblatt's paper.

REFERENCES

1. Busacker, R. G., and Saaty, T. L. Finite Graphs and Networks. McGraw-Hill, New York, 1965.

2. Gibbs, N. E. A Cycle Generation Algorithm for Finite Undirected Linear Graphs, JACM, 16, 4 (Oct. 1969), 564-568.

3. Gotlieb, C. C., and Corneil, D. G. Algorithms for Finding a Fundamental Set of Cycles for an Undirected Linear Graph. Comm. ACM 10, 12 (Dec. 1967), 780-783.

4. Paton, Keith. An Algorithm for Finding a Fundamental Set of Cycles of a Graph. Comm. ACM 12, 9 (Sept. 1969), 514-518.

5. Roberts, S. M., and Flores, Benito. Systematic Generation of Hamiltonian Circuits. Comm. ACM 9, 9 (Sept. 1966), 690-694.

6. 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.).

7. Schurmann, A. The Application of Graphs to the Analysis of Distribution of Loops in a Program. Inform. Contr. 1 (1964), 275-282.

8. Tiernan, S. C. An Efficient Search Algorithm to Find the Elementary Circuits of a Graph. Comm. ACM 13, 12 (Dec. 1970), 722-727.

9. Weinblatt, H. A New Search Algorithm for Finding the Simple Cycles of a Finite Directed Graph, JACM, 19, 1, (Jan. 1972), 43-56.

10. Welch, J. T. A Mechanical Analysis of the Cyclic Structure of Undirected Linear Graphs. JACM 13, 2 (Apr. 1966), 205-210.

APPENDIX I

Simple Bounds on the Number of Cycles in the Complete Digraph

$$N_c = n! \sum_{m=2}^{n} \frac{1}{m((n-m)!)}$$

First we derive a least upper bound on

$$S_n = \sum_{m=2}^{n} \frac{1}{m((n-m)!)}$$

Consider the difference $S_n - S_{n+1}$

First S_{n+1} has an additional term

$$T_1' = \frac{1}{2((n+1-2)!)}$$

Consider the m^{th} term T_m of S_n and the $m+1$ term T_{m+1}' of S_{n+1}

$$T_m = \frac{1}{m((n-m)!)}$$

$$T_{m+1}' = \frac{1}{(m+1)((n+1-(m+1))!)} = \frac{1}{(m+1)((n-m)!)}$$

Now $$\frac{1}{m((n-m)!)} > \frac{1}{(m+1)((n-m)!)}$$ and

$$T_m > T_{m+1}'$$

Since

$$T_{n-1} + T_n = \frac{1}{n-1} + \frac{1}{n} = \frac{2n-1}{n(n-1)} = \frac{2n^2+n-1}{n(n+1)(n-1)}$$

$$T_n' + T_{n+1}' = \frac{1}{n} + \frac{1}{n+1} = \frac{2n+1}{n(n+1)} = \frac{2n^2-n-1}{n(n+1)(n-1)}$$

Then

$$(T_{n-1}+T_n) - (T'_n+T'_{n+1}) = \frac{2}{n^2-1} \geq T'_1$$

Now

$$\frac{2}{n^2-1} \geq \frac{1}{2(n-1)!}$$

Because

$$\frac{4}{n+1} \geq \frac{1}{(n-2)!} \qquad \text{for} \quad n=3,4,\ldots$$

and thus $S_{n+1} < S_n$ so that S_3 is a least upper bound and $S_2=1/2$

and $S_3=5/6$ implies,

$$N_c < \frac{5}{6}\, n! \quad n \geq 2$$

To derive a simple lower bound on N_c note that the last two terms are the largest.

$$\frac{n!}{(n-1)} + \frac{n!}{n} = n((n-2)!) + (n-1)!$$

Since $n((n-2)!) = ((n-1)+1)((n-2)!) = (n-1)! + (n-2)!$

$$\frac{n!}{(n-1)} + \frac{n!}{n} = 2((n-1)!) + (n-2)! > 2((n-1)!)$$

and $2((n-1)!) < N_c \qquad n=3,4,\ldots$

Thus $2((n-1)!) < n! \sum_{m=2}^{n} \frac{1}{m((n-m)!)} \leq \frac{5}{6}\, n! \quad , \; n > 2 \quad .$