SIAM J. COMPUT.
Vol. 4, No. 1, March 1975

FINDING ALL THE ELEMENTARY CIRCUITS OF A DIRECTED GRAPH*

DONALD B. JOHNSON†

Abstract. An algorithm is presented which finds all the elementary circuits of a directed graph in time bounded by $O((n + e)(c + 1))$ and space bounded by $O(n + e)$, where there are n vertices, e edges and c elementary circuits in the graph. The algorithm resembles algorithms by Tiernan and Tarjan, but is faster because it considers each edge at most twice between any one circuit and the next in the output sequence.

Key words. algorithm, circuit, cycle, enumeration, digraph, graph

1. Introduction. Broadly speaking, there are two enumeration problems on sets of objects. The one, which we call *counting*, is determining how many objects there are in the set. The other, which we call *finding*, is the construction of every object in the set exactly once. Indeed, objects may always be counted by finding them if a method to do so is at hand. But knowing the count is usually of little aid in finding the objects.

We give an algorithm for finding the elementary circuits of a directed graph which is faster in the worst case than algorithms previously known. As far as we know, it is also the fastest method known for the general enumeration problem as well (see [1, p. 226]). Specific counting problems are, of course, solved. For example, there are exactly

$$\sum_{i=1}^{n-1} \binom{n}{n-i+1} (n-i)!$$

elementary circuits in a complete directed graph with n vertices. Thus the number of elementary circuits in a directed graph can grow faster with n than the exponential 2^n. So it is clear that our algorithm, which has a time bound of $O((n + e)(c + 1))$ on any graph with n vertices, e edges and c elementary circuits, is feasible for a substantially larger class of problems than the best algorithms previously known [2], [3], which realize a time bound of $O(n \cdot e(c + 1))$.

A *directed graph* $G = (V, E)$ consists of a nonempty and finite set of vertices V and a set E of ordered pairs of distinct vertices called *edges*. There are n vertices and e edges in G. A *path* in G is a sequence of vertices $p_{vu} = (v = v_1, v_2, \cdots, v_k = u)$ such that $(v_i, v_{i+1}) \in E$ for $1 \leqq i < k$. A *circuit* is a path in which the first and last vertices are identical. A path is *elementary* if no vertex appears twice. A circuit is elementary if no vertex but the first and last appears twice. Two elementary circuits are distinct if one is not a cyclic permutation of the other. There are c distinct elementary circuits in G. Our definitions exclude graphs with loops (edges of the form (v, v)) and multiple edges between the same vertices. It is obvious that for any circuit q_{vv}, there exists a vertex u such that q_{vv} is composed of a path p_{vu} followed by edge (u, v). If q_{vv} is elementary, then p_{vu} is also elementary.

* Received by the editors December 10, 1973, and in final revised form June 10, 1974.

† Computer Science Department, Pennsylvania State University, University Park, Pennsylvania 16802.

F is a *subgraph of* G *induced by* W if $W \subseteq V$ and $F = (W, \{(u, v) | u, v \in W$ and $(u, v) \in E\})$. An induced subgraph F is a (maximal) *strong component of* G if for all $u, v \in W$ there exist paths p_{uv} and p_{vu} and this property holds for no subgraph of G induced by a vertex set $\overline{W}$ such that $W \subset \overline{W} \subseteq V$.

The literature contains several algorithms which find the elementary circuits of any direct graph. In the algorithms of Tiernan [4] and of Weinblatt [5], time exponential in the size of the graph may elapse between the output of one circuit and the next [2]. Tarjan [2] presents a variation of Tiernan's algorithm in which at most $O(n \cdot e)$ time elapses between the output of any two circuits in sequence, giving a bound of $O(n \cdot e(c + 1))$ for the running time of the algorithm on an entire graph in the worst case. Ehrenfeucht, Fosdick, and Osterweil [3] give a similar algorithm which realizes the same bound.

In the case of Tarjan's algorithm, the worst-case time bound is realized, for instance, on the graph shown in Fig. 1. We assume that the algorithm begins with vertex 1 and, in any search from vertices 1 through $k + 1$, it visits vertices $k + 2$ through $2k + 1$ before a first visit to vertex $2k + 2$. In the course of finding each of the k elementary circuits which contain vertex 1, the subgraph on vertices $2k + 2$ through $3k + 3$ will be explored k times, once for each of the vertices $k + 2$ through $2k + 1$. Thus exploration from vertex 1 alone consumes $O(k^3)$ time. Since there are exactly $3k$ elementary circuits in the entire graph, the running time is at least $O(n \cdot e(c + 1))$.

The worst-case time bound for Tarjan's algorithm is also realized on the graph in Fig. 2. Assuming a start at vertex 1, the algorithm takes $O(k)$ time to find the one elementary circuit of the graph. Then the fruitless searches from vertices 2 through k take $O(k^2)$ time, which is $O(n \cdot e(c + 1))$. So we see that there are two ways in which the time bound is realized in Tarjan's algorithm. One is through repeated fruitless searching of a subgraph while seeking circuits with a certain least vertex; the other is through fruitless searches from many vertices which are least vertices in no elementary circuit.

The graphs of Figs. 1 and 2 are strongly connected, and their undirected versions are biconnected. On such graphs, obvious preprocessing techniques, such as reducing the graph to its strong components (as Weinblatt does [5]) or to components which in addition to strong connectivity have biconnected undirected versions, do not improve the performance of Tarjan's algorithm. Stronger techniques are needed to get a better asymptotic running time in the worst case.

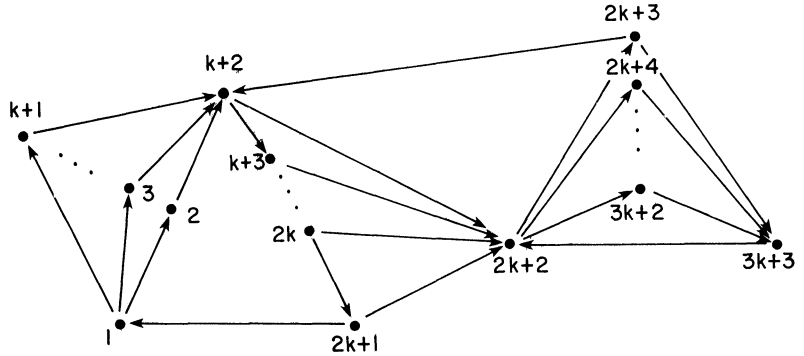

FIG. 1. *A worst-case example for Tarjan's algorithm*

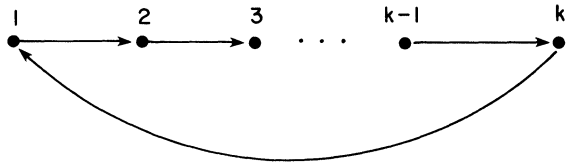

FIG. 2. *A second worst-case example for Tarjan's algorithm*

2. The algorithm. In our algorithm, the time consumed between the output of two consecutive circuits as well as before the first and after the last circuits never exceeds the size of the graph, $O(n + e)$. We employ the basic notion of Tiernan's algorithm. Elementary circuits are constructed from a root vertex s in the subgraph induced by s and vertices "larger than s" in some ordering of the vertices. Thus the output is grouped according to least vertices of the circuits.

To avoid duplicating circuits, a vertex v is *blocked* when it is added to some elementary path beginning in s. It stays blocked as long as every path from v to s intersects the current elementary path at a vertex other than s. Furthermore, a vertex does not become a root vertex for constructing elementary paths unless it is the least vertex in at least one elementary circuit. These two features avoid much of the fruitless searching of Tiernan's, Weinblatt's and Tarjan's algorithms and of the algorithm of Ehrenfeucht, Fosdick, and Osterweil.

The algorithm accepts a graph G represented by an adjacency structure A_G composed of an adjacency list $A_G(v)$ for each $v \in V$. The list $A_G(v)$ contains u if and only if edge $(v, u) \in E$. The algorithm assumes that vertices are represented by integers from 1 to n.

The algorithm proceeds by building elementary paths from s. The vertices of the current elementary path are kept on a stack. A vertex is appended to an elementary path by a call to the procedure CIRCUIT and is deleted upon return from this call. When a vertex v is appended to a path it is blocked by setting blocked (v) = true, so that v cannot be used twice on the same path. Upon return from the call which blocks v, however, v is not necessarily unblocked. Unblocking is always delayed sufficiently so that any two unblockings of v are separated by either an output of a new circuit or a return to the main procedure.

CIRCUIT-FINDING ALGORITHM

```
begin
    integer list array A_K(n), B(n); logical array blocked (n); integer s;
    logical procedure CIRCUIT (integer value v);
        begin logical f;
            procedure UNBLOCK (integer value u);
                begin
                    blocked (u) := false;
                    for w∈B(u) do
                        begin
                            delete w from B(u);
                            if blocked(w) then UNBLOCK(w);
                        end
                end UNBLOCK;
            f := false;
```

```
            stack v;
            blocked(v) := true;
    L1:     for w∈A_K(v) do
                if w=s then
                    begin
                        output circuit composed of stack followed by s;
                        f := true;
                    end
                else if ¬blocked(w) then
                        if CIRCUIT(w) then f := true;
    L2:     if f then UNBLOCK(v)
            else for w∈A_K(v) do
                    if v∉B(w) then put v on B(w);
            unstack v;
            CIRCUIT := f;
        end CIRCUIT;
    empty stack;
    s := 1;
    while s < n do
        begin
            A_K := adjacency structure of strong component K with least
                   vertex in subgraph of G induced by {s, s+1, ···, n};
            if A_K ≠ ∅ then
                begin
                    s := least vertex in V_K;
                    for i∈V_K do
                        begin
                            blocked(i) := false;
                            B(i) := ∅;
                        end;
    L3:             dummy := CIRCUIT(s);
                    s := s+1;
                end
            else s := n;
        end
end;
```

The correctness of the algorithm depends on no vertex remaining blocked when there is a path from the vertex to s which intersects the stack only at s. On the other hand, the bound on running time depends on all vertices remaining blocked as long as possible, consistent with the requirements of correctness. The following lemma establishes these properties. It will be seen that the B-lists are used to remember information obtained from searches of portions of the graph which do not yield an elementary circuit. The procedure UNBLOCK has the property that if there is a call UNBLOCK(x) and vertex y is on list $B(x)$, then there will be a call UNBLOCK(y) following which blocked(y) will be false.

LEMMA 1. *At* L2, *for any vertex* $x \neq s$, *there is a call* UNBLOCK(v) *which sets* blocked(x) = false *if and only if*

(i) *there is a path, containing v, from x to s on which only v and s are on the stack, and*

(ii) *there is no path from x to s on which only s is on the stack.*

Proof. Assume, to the contrary, that there is an execution of L2 at which the lemma first fails and that the lemma fails for no vertex before it fails for vertex y. Two cases are possible under this assumption.

Case 1. Suppose that the path conditions, (i) and (ii), hold for y at L2, but blocked(y) is not set false. Because there is an edge (v, z) on the path from y to s, f is true at L2. This fact is immediate if $z = s$. If $z \neq s$, it follows from our assumption that the lemma holds for z before the return from the call CIRCUIT(z). Thus there is a call UNBLOCK(v) and, without loss of generality, a path $(y = v_1, v_2, \cdots, v_k = v)$ on which only v is on the stack and only y is not unblocked as a result of the call UNBLOCK(v) at L2. But when y was last blocked, y was on the stack. Since y remained blocked when y was removed from the stack, y was put on list $B(v_2)$. So there was a call UNBLOCK(y), a contradiction.

Case 2. Suppose that there is a call UNBLOCK(v) at L2 and that blocked(y) is set false, but that either (i) or (ii) is not satisfied. It cannot be that $v = s$ because it is clear that the lemma holds when only $v = s$ is on the stack, and s cannot be stacked more than once. Since f is true there is an edge (v, z) such that either $z = s$ or f was set true when the call CIRCUIT(z) returned. It follows from our assumption that, when f was set true, there was a path from z to s on which only s was on the stack. It may be that several calls to CIRCUIT occur after f is set true and before the current call UNBLOCK(v). In any event, the current stack (when the call UNBLOCK(v) occurs) is identical to the stack when f was set true, so there is a path $(v, z, \cdots, s)$ on which only v and s are on the stack. Since v is on the stack, (i) and (ii) would be satisfied if $y = v$.

So $y \neq v$, and there is some vertex t which is unblocked before y is unblocked such that y is on $B(t)$. By assumption, there is a path from t to s on which only v and s are on the stack. Furthermore, when y was last put on $B(t)$, blocked(t) was true and y was on the stack. But blocked(t) has to remain true until the current call UNBLOCK(v). Otherwise y would have been removed from $B(t)$. Since y must have been unstacked after y was put on $B(t)$, there was some execution of L2 where the stack was a prefix of the current stack, (i) and (ii) held for t, and blocked(t) was not set false. But by assumption, the lemma did not fail then for t. From this contradiction, we find that Case 2 is also impossible. □

COROLLARY 1. *The algorithm outputs only elementary circuits.*

Proof. Certainly only circuits are output. By Lemma 1, a vertex is only unblocked if it will be off the stack before any call to CIRCUIT can occur. Thus no vertex can be repeated on the stack. □

LEMMA 2. *The algorithm outputs every elementary circuit exactly once.*

Proof. No circuit is output more than once since, for any stack $(s = v_1, v_2, \cdots, v_k)$ with v_k on top, once v_k is removed the same stack cannot reoccur.

Let $(v_1, v_2, \cdots, v_l, v_1)$ be an elementary circuit such that $v_1 \leqq v_i$, $1 \leqq i \leqq l$. A first call CIRCUIT(v_1) will eventually occur at L3 since there is a strong component with least vertex v_1. Since no vertex is blocked when this first call occurs, it follows by induction using Lemma 1 that whenever the stack is $(s = v_1, v_2, \cdots, v_i)$ for $i < l$, the stack will later be $(s = v_1, v_2, \cdots, v_{i+1})$. Thus every elementary circuit is output. □

The foregoing results show that the algorithm does indeed find all the elementary circuits of a directed graph. The bound on running time follows from the next lemma.

LEMMA 3. *At most $O(n + e)$ time can elapse in a call at* L3 *to* CIRCUIT *before either the call returns or a circuit is output.*

Proof. First we show that no vertex can be unblocked twice in succession unless a circuit is output. Then we show that no more than $O(n + e)$ time can elapse before some vertex is unblocked a second time.

Suppose a circuit is output and then some vertex y is unblocked. By Lemma 1, as soon as v is unstacked there is a path $(y = v_1, v_2, \cdots, v_k = s)$ on which only s is on the stack. Let some vertex v_i, $1 \leqq i < k$, be the first vertex on this path to be put on the stack again. We see by induction on the execution of the algorithm that eventually the stack will be $(s, \cdots, v_i, v_{i+1}, \cdots, v_{k-1})$ and a new circuit output. Until the new circuit is output, no vertex on the path will be unstacked. Thus no vertex can be unblocked more than once before a circuit is output.

Charge a unit of cost to a vertex if it is an argument to a procedure call and a unit of cost to an edge if consideration of this edge by the **for** loop at L1 in CIRCUIT does not result in a procedure call. The cost of all work in the procedure CIRCUIT will be bounded by a constant times the number of units charged. For any vertex x, calls to CIRCUIT and UNBLOCK must alternate. Consequently, no more than three units can be charged to each vertex before some vertex is unblocked twice. As to edge charges, let some edge originate in vertex x. A unit may be charged to this edge only when blocked(x) is true and, once a unit is charged, x must be unblocked and blocked again before a second unit can be charged to the same edge. It follows that at most two units can be charged to any edge before some vertex is unblocked twice. □

COROLLARY 2. *The algorithm runs in $O((n + e)(c + 1))$ time and uses $O(n + e)$ storage space plus the space used for output.*

Proof. The time bound follows directly from Lemma 3 and a known algorithm [6] for finding strong components in $O(n + e)$ time. The space bound is immediate from the observation that no vertex appears more than once on any B-list. □

3. Discussion of running time. We have shown that in the worst case, our algorithm is asymptotically faster than algorithms previously known. With respect to Tarjan's algorithm, a stronger statement can be made. There is a constant factor which bounds how much slower our algorithm can be compared to his on any graph, provided the same adjacency structure is used as input to both algorithms. Such a constant, of course, is implementation dependent. Its existence follows from two facts. First, the time spent by our algorithm in finding the strong component with least vertex $s \geqq k$ is of no greater order than the search in Tarjan's algorithm for circuits with least vertex k, for $1 \leqq k \leqq n$. Second, if the calls to UNBLOCK are ignored, on identical adjacency structures the sequence of edge explorations generated by our algorithm is embedded in the sequence generated by Tarjan's. But for every edge in the sequence, for our algorithm there can occur at most one call to UNBLOCK. Therefore, since the search time in each algorithm is related by constant factors to the number of edge explorations, the effort spent by our algorithm in finding circuits from a given base vertex, s, is bounded by a

constant factor times the effort expended by Tarjan's algorithm for the corresponding search on the same adjacency structure for any graph.

Experimental results are shown in Tables 1 and 2. The algorithms were implemented in ALGOL W [7] and were run on an IBM 370/168 with virtual address hardware inoperative. The benefit predicted for our algorithm on worst cases is apparent in the results in Table 1. Table 2 shows superior performance by our algorithm on complete graphs as well. Although only a constant factor is involved, this second result is somewhat surprising since on complete graphs, both algorithms make the same number of edge explorations. The result, however, appears to be explained by two features of Tarjan's algorithm. He maintains two vertex stacks and tests in the innermost loop for elimination of vertices less than s. If his algorithm were redesigned to correct these problems, the analysis of the

TABLE 1
Running times on the family of graphs of Fig. 1

Number of vertices	Number of circuits	Running time on IBM 370/168, seconds[1]		T_T/T_J
		T_J (Johnson's algorithm)	T_T (Tarjan's algorithm)	
5	15	.03	.06	2
10	30	.11	.27	2.5
20	60	.32	1.67	5.2
40	120	1.17	11.51	9.8
60	180	2.61	36.89	14.1
80	240	4.46	86.66	19.4

[1] Timer resolution 1/60 second. Because running times fluctuate with system load, all data shown were taken from one computer run. Results are averages of two times rounded to the second decimal place.

TABLE 2
Running times on complete directed graphs

Number of vertices	Number of circuits	Running time on IBM 370/168, seconds[2]		T_T/T_J
		T_J (Johnson's algorithm)	T_T (Tarjan's algorithm)	
2	1	0	0	—
3	5	0	0	—
4	20	.02	0	—
5	84	.02	.02	1
6	409	.07	.08	1.1
7	2365	.35	.51	1.5
8	16064	2.43	3.63	1.5
9	125664	20.17	30.13	1.5

[2] Timer resolution 1/60 second. Because running times fluctuate with system load, all data shown were taken from one computer run. Results are averages of two times rounded to the second decimal place.

preceding paragraph would still hold. In both tests, the space bound of $O(n + e)$ was confirmed.

4. Conclusions. The algorithm we have shown is faster asymptotically in the worst case than algorithms previously known. The algorithm appears particularly suited for general use because of the stronger property, which we have shown in relation to Tarjan's algorithm, of being never slower on any graph by more than a constant factor. In fact, in the tests run, our algorithm was always faster except on trivially small graphs.

Acknowledgment. Credit is due to the referees for suggesting that, for comparative tests, a family of graphs be found in which the members are strongly connected and whose undirected versions are biconnected, and for a suggestion that led to an improvement, by a constant factor, in running time.

REFERENCES

[1] F. HARARY AND E. PALMER, *Graphical Enumeration*, Academic Press, New York, 1973.

[2] R. TARJAN, *Enumeration of the elementary circuits of a directed graph*, this Journal, 2 (1973), pp. 211–216.

[3] A. EHRENFEUCHT, L. FOSDICK AND L. OSTERWEIL, *An algorithm for finding the elementary circuits of a directed graph*, Tech. Rep. CU-CS-024-23, Dept. of Computer Sci., Univ. of Colorado, Boulder, 1973.

[4] J. C. TIERNAN, *An efficient search algorithm to find the elementary circuits of a graph*, Comm. ACM, 13 (1970), pp. 722–726.

[5] H. WEINBLATT, *A new search algorithm for finding the simple cycles of a finite directed graph*, J. Assoc. Comput. Mach., 19 (1972), pp. 43–56.

[6] R. TARJAN, *Depth-first search and linear graph algorithms*, this Journal, 1 (1972), pp. 146–160.

[7] R. L. SITES, *Algol W reference manual*, Tech. Rep. STAN-CS-71-230, Computer Sci. Dept., Stanford Univ., Stanford, Calif., 1971.