BOUNDS ON BACKTRACK ALGORITHMS FOR LISTING CYCLES, PATHS, AND SPANNING TREES

by

Ronald C. Read and R. Endre Tarjan

Memorandum No. ERL-M433

December 1973

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

BOUNDS ON BACKTRACK ALGORITHMS FOR LISTING CYCLES, PATHS, AND SPANNING TREES

by

Ronald C. Read*
Department of Combinatorics & Optimization
University of Waterloo
Waterloo, Canada.

R. Endre Tarjan†
Computer Science Division
University of California
Berkeley, California

December, 1973.

* Research partially supported by the National Research Council of Canada, Grant # A8142.

† Research partially supported by the National Science Foundation, Contract Number GJ-35604X, and by a Miller Research Fellowship.

ABSTRACT

Backtrack algorithms for listing certain kinds of subgraphs of a graph are described and analysed. Included are algorithms for enumerating spanning trees, cycles, simple cycles, and other kinds of paths. The algorithms have O(V+E) space requirements and O(V+E+EN) time requirements, if the problem graph has V vertices, E edges, and N subgraphs of the type to be listed.

Keywords: algorithm, circuit, cycle, enumeration, graph, path, spanning tree, tree.

BOUNDS ON BACKTRACK ALGORITHMS FOR LISTING CYCLES, PATHS, AND SPANNING TREES

by

Ronald C. Read and R. Endre Tarjan

Introduction

Certain applications of graph theory require the counting or listing of the set of subgraphs of a graph which satisfy some particular property. Examples include calculation of electrical circuit parameters by using spanning trees [1], studying program flow by using the cycles in a program flow graph [2], and organizing information according to the cliques of an associated data graph [3]. One algorithmic technique, backtracking, is particularly suitable for listing the kinds of subgraphs which occur in some of these applications.

We may enumerate all subsets of a set S by choosing some ordering for the elements of S and examining the elements in order. When we examine an element, we decide whether to include it or not in the subset we are constructing. After we decide whether to include the last element, we list the set we have just constructed, then we change our decision about the last element and list a new set. Whenever we have tried both including and excluding an element, we back up to the previous element, change our decision, and move forward again. This process is called <u>backtracking</u> [4].

By listing all subsets of the edges of a given graph using backtracking and testing each subset to see if it satisfies a desired property, we can list all subgraphs which have the desired property. However, very few of the possible subgraphs may satisfy the property. If we can predict as we go along whether the subgraph we are constructing can be completed to form a desired one, then we can restrict the back-tracking process, exploring only fruitful possibilities, and thus list the desired subgraphs much more efficiently than if we do notrestrict the backtracking. This paper presents and analyzes efficient backtrack algorithms for listing spanning trees, cycles, simple cycles, and certain other kinds of paths.

Definitions:

A graph $G = (\mathcal{V},\mathcal{E})$ is a collection of vertices $\mathcal{V}$ and edges $\mathcal{E}$. V denotes the number of vertices and E the number of edges. The edges may be either unordered pairs of distinct vertices (the graph is undirected) or ordered pairs of distinct vertices (the graph is directed). An edge is denoted by $e = (v,w)$; v and w are adjacent, v and w are incident to (v,w). Graphs do not contain loops (edges of the form (v,v))or multiple edges, though it is easy to modify the algorithms to allow loops and multiple edges.

A sequence of distinct edges $p = (v_1,v_2), (v_2,v_3), \ldots, (v_{n-1},v_n)$ in G is called a path from v_1 to v_n. If $v_1 = v_n$, p is a cycle. If $v_1, v_2, \ldots, v_n$ are all distinct, p is a simple path. If $v_1 = v_n$ and $v_1, v_2, \ldots, v_{n-1}$ are distinct, p is a simple cycle. By convention, a path may contain no edges, but a cycle must contain at least two edges. Two cycles which are cyclic permutations of each other are regarded as the same cycle. A graph $G' = (\mathcal{V}', \mathcal{E}')$ is a subgraph of G if $\mathcal{V}' \subseteq \mathcal{V}$ and $\mathcal{E}' \subseteq \mathcal{E}$. An undirected graph is connected if there is a path between every distinct pair of vertices. The maximal connected subgraphs of a graph are its connected components. A directed graph is strongly connected if for any two distinct vertices v and w, there is a path from v to w. An edge e of an undirected graph is a bridge if there are two vertices v and w which have at least one path joining them and every path joining them contains e. A tree is a connected undirected graph which contains no cycles. A spanning tree of an undirected graph G is a subgraph which is a tree and which contains all the vertices of G. A complete graph is a graph with any two distinct vertices joined by an edge. A clique of a graph G is a maximal complete subgraph of G. A directed, rooted tree is a directed graph with one vertex, the root, having no edges leading to it, all other vertices having one edge leading to them, and no cycles. If there is a path from v to w in a directed rooted tree, then v is called an ancestor of w and w is a descendant of v.

Listing Spanning Trees

The problem of listing all spanning trees of an undirected graph has an application in the solution of linear electrical networks [1], and several algorithms have been published [1,5-9]. Many of them use backtracking restricted in some way. If T is the number of spanning trees of the graph, we would like to have an algorithm which runs in a time bound polynomial in V,E, and T, and which requires as little storage as possible. Suppose we generate all subsets of the edges of a graph by backtracking, and list those which give spanning trees. This simple algorithm was apparently first suggested by Feussner [7]. Some of the subsets are spanning trees, but others are not. In fact, if the original graph is a tree, such an unrestricted backtracking algorithm will require at least $k2^E$ time (for some constant k) to find one spanning tree! Thus, we must restrict the backtracking to get an efficient algorithm.

One of the algorithms in the literature, McIlroy's [5], is a minor variation of unrestricted backtracking. McIlroy's algorithm works vertex by vertex, "growing" a spanning tree by adding one vertex at a time to it. At a given step the algorithm picks a vertex adjacent to the current tree and connects it to the tree using some edge. When the algorithm backtracks to this vertex, it deletes the previous connecting edge from the graph and picks another connecting edge. When no more possible connecting edges remain to be chosen, the algorithm picks another vertex adjacent to the tree and connects it. When no remaining vertices are adjacent to the tree, the algorithm backtracks. This algorithm has a partial check against looking for spanning trees in a disconnected graph, but on a graph such as that in Figure 1 it will require $k2^E$ time to find one spanning tree, if it starts at vertex 1, and it is not an efficient algorithm in the worst case.

However, there is one algorithm in the literature which is very efficient. The algorithm was described informally by Minty [6]; here we give a version of the algorithm in Algol-like notation and a worst-case running time analysis. The algorithm is based on two observations which restrict the necessary backtracking. First, any bridge of a graph must be in all its spanning trees. Thus, at any time during backtracking when we are deciding whether to include or delete an edge which is a bridge, we may automatically include it. Second, any edge which forms a cycle with edges already included in the spanning tree must not be included as a spanning tree edge. These two checks give rise to the following algorithm for listing spanning trees:

```
procedure SPAN; begin
     procedure REC; begin
          declare B, L set variables local to procedure REC;
     S1:  find all bridges B which are not tree edges;
          add all edges in B to current partial spanning tree;
     S2:  find all edges L not in partial tree joining vertices already
                 connected in partial tree;
          delete all edges L from graph;
          if all edges in tree then output spanning tree else begin
               let e be an edge not in tree;
               add e to tree;
               REC;
               delete e from tree and from graph;
               REC;
               remove all edges in B from tree;
               add e and all edges in L to graph;
          end;
     end;
     initialize current partial spanning tree to the empty set;
S3:  test to see that graph is connected;
     if graph is not connected then output no trees else REC;
end;
```

This algorithm is very easy to program. Step S1 may be implemented using depth-first search; all the bridges of a graph may be found in O(V+E) time using such a search [10,11,12]. Step S2 may be carried out by using any search process [11] to find the connected components of the partially constructed spanning tree, then labelling the vertices of each component with distinguishing numbers, and finally adding to L each edge which joins two vertices having the same number. The total time for these operations is O(V+E). Step S3 may also be carried out in O(V+E) time using a search [11].

If the problem graph is not connected, it contains no spanning trees, and algorithm SPAN will indicate this in O(V+E) time after step S3 is completed. If the problem graph is connected, then $E \geq V-1$ and a single call on REC requires O(E) time plus (possibly) time for two nested calls on REC. Each call on REC gives rise either to a spanning tree or to two nested calls on REC. Thus, the nested calls on REC may be represented as a binary tree with the bottom-most calls corresponding to spanning trees. It follows that the number of calls on REC is O(T), where T is the number of spanning trees. The total running time of SPAN is thus O(V+E+ET). SPAN requries O(V+E) storage space plus (possibly) storage space for the spanning trees. If the trees are used as they are generated, SPAN requires O(V+E) total storage space.

Algorithm SPAN is almost as efficient as is theoretically possible. Any spanning tree algorithm must look at the entire problem graph and must list all spanning trees. Thus, any spanning tree algorithm requires at least k(V+E+VT) time for some constant k. Ignoring constant factors, SPAN is within a factor of $\frac{E}{V}$ of being as efficient as theoretically possible.

We believe that it is possible to construct an algorithm with a time bound closer to the theoretical lower limit by using a reference tree idea [13] coupled with a good algorithm for computing equivalence relations [14-16], but the improvement afforded by such an algorithm would be minimal for two reasons. First, the algorithm would be significantly more complicated than SPAN (the constant factor in the running time would be much greater). Second, if E/V is large, the number of spanning trees is very large, as demonstrated in the theorem below.

Theorem 1:

A connected graph G with V vertices and E edges has at least 2^t spanning trees, where $t = \left\lceil \frac{-1 + \sqrt{1+8(E-V+1)}}{2} \right\rceil$.

Proof:

Pick any particular spanning tree J of G and delete all the edges of J from G to form a graph G'. Let J' be a graph consisting of a set of trees, one spanning each connected component of G'. If J' contains t edges and the connected components of G' have $n_1, n_2, \ldots, n_k$ vertices, then

$$t = \sum_{i=1}^{k} (n_i-1) \quad \text{and} \quad \sum_{i=1}^{k} \frac{n_i(n_i-1)}{2} \geq E - V + 1.$$

This implies $t(t+1) = \left(\sum_{i=1}^{k} (n_i-1)\right)^2 + \sum_{i=1}^{k} (n_i-1)$

$$\geq \sum_{i=1}^{k} [(n_i-1)^2 + (n_i-1)] = \sum_{i=1}^{k} n_i(n_i-1) \geq 2(E - V + 1).$$

Thus, $$t \geq \left\lceil \frac{-1 + \sqrt{1 + 8(E-V+1)}}{2} \right\rceil .$$

By combining each subset of the edges of $\mathcal{J}'$ with an appropriate subset of the edges of $\mathcal{J}$, we may form 2^t different spanning trees of G. This proves the theorem.

Corollary 2:

SPAN's worst case running time is within a factor of $k \log (T+2) \log\log(T+4)$ of best possible, for some constant k.

Proof:

If graph G is disconnected or $E < 2V$, SPAN's running time is best possible to within a constant factor. If G is connected and $E \geq 2V$, $T \geq 2^t$ where

$$t = \left\lceil \frac{-1 + \sqrt{1 + 8(E-V+1)}}{2} \right\rceil .$$

It follows that $\log T(\log T+1) \geq 2(E-V+1)$. Thus, $(\log T)^2 \geq E-V+1$, $\log T \geq \sqrt{E-V+1}$, and $\log\log T \geq \frac{1}{2} \log V$. Since $T \leq V^{V-2}$, $\log T \leq V \log V$. Combining, we have

$$\log T \log\log T \geq \frac{1}{2} \log T \log V \geq \frac{1}{2} \frac{(\log T)^2}{V} \geq \frac{1}{2} \frac{E-V+1}{V} \geq \frac{1}{4} \frac{E}{V} .$$

The corollary follows.

Theorem 3:

Given any V and E, there is a connected graph with V vertices, E edges, and fewer than t^t spanning trees, where

$$t = \left\lceil \frac{3 + \sqrt{9 + 8(E-V)}}{2} \right\rceil .$$

Proof:

Construct a graph of V vertices and E edges consisting of a spanning tree plus edges between vertices in a fixed set of t vertices. This graph has fewer than t^t spanning trees. We can construct such a graph if $\frac{t(t-1)}{2} + V - t \geq E$; that is, if $t^2 - 3t - 2(E-V) \geq 0$. But,

$$t \geq \left\lceil \frac{3 + \sqrt{9 + 8(E-V)}}{2} \right\rceil$$

guarantees this.

Theorem 3 shows that the bound in Theorem 1 is reasonably tight.

Listing Simple Cycles

Backtracking is also very useful for listing the simple cycles (and certain other kinds of paths) in an undirected or a directed graph. A list of simple cycles is useful in optimizing computer programs, and several published papers contain algorithms [17-27] . The two most common approaches are to use some kind of restricted backtracking or to represent cycles as elements in a vector space whose basis is a set of fundamental cycles. Prabhaker and Deo [24] discuss these and other methods.

The vector space approach has two significant drawbacks. First, it is only applicable to undirected graphs. Second, for each of the algorithms in [22,25] it is possible to construct a graph with a small number of cycles for which the given algorithm requires an exponential amount of time to list the simple cycles. This second drawback (inefficiency) is shared by the algorithms described in [17,18,20,26,27]. Reference [19] contains bad example graphs for the algorithms of

Tiernan [17] and Weinblatt [18]; bad examples for the other algorithms are easily constructed.

The three reasonably efficient algorithms in the literature [19, 21, 23] all use the same type of backtracking, though restricted in slightly different ways. Suppose we somehow number the vertices of the problem graph from 1 to V. Then we can pick the smallest numbered vertex on a cycle to be a unique "start" vertex of the cycle. For a fixed start vertex, we choose an edge leading from the vertex. This starts a path in the problem graph. To extend the path, we pick an edge leading from the last vertex on the path. We do not allow the path to intersect itself or to contain vertices smaller than the start vertex. If the last vertex on the path has an edge leading to the start vertex, we output a cycle. When we have tried all possible ways of extending a path, we back up, delete the last edge on the path, and try another possibility. To ennumerate all simple cycles, we try this backtracking procedure with each vertex as a starting vertex. Tiernan was the originator of this algorithm, which is sometimes very inefficient because many ways of extending a path may not lead to a simple cycle [19].

However, it is possible to put restrictions on the backtracking and get an efficient algorithm. The algorithms in [19, 21, 23] are all of this type. If the problem graph has C simple cycles, Ehrenfeucht, Fosdick, and Osterweil's algorithm [21] has an $O(VE(C+1))$ time bound, Tarjan's [19] has an $O(VE(C+1))$ time bound, and Johnson's [23] has an $O(V + E + EC)$ time bound. All three algorithms have $O(V+E)$ space requirements in addition to storage space for the output. A new algorithm with an

O(V+E+EC) time bound appears below. This algorithm was developed independently of Johnson's and rivals it in simplicity and theoretical efficiency.

To improve Tiernan's essentially unrestricted backtracking procedure, we need a way to select possible cycle starting vertices and a way to restrict the backtracking. Suppose we explore the problem graph using depth-first search [10], numbering the vertices from 1 to V as we reach them during the search. The search divides the edges of the graph into four sets: a set of edges forming a directed rooted tree; a set of edges from descendants to ancestors in the tree, called _cycle arcs_; a set of edges from ancestors to descendants in the tree, called _forward arcs_; and a set of edges joining unrelated vertices in the spanning tree, called _cross arcs_. The tree spans all vertices reachable from the start vertex, which is the root of the tree. Tree edges and forward arcs run from smaller to larger numbered vertices; cycle arcs and cross arcs run from larger to smaller numbered vertices [10]. Each cycle arc is an edge of at least one simple cycle, and each cycle must end in a cycle arc. This follows from results in [28]. Thus, the set of cycle-starting vertices is exactly the set of vertices with entering cycle arcs, a set which we can determine in O(V+E) time using depth-first search [10]. We may also use this search to divide the graph into strongly connected components, each of which may be processed separately for cycles.

For each cycle start vertex, we carry out a backtracking procedure to build up a simple path which may be extended into a simple cycle. After adding a new vertex to the current path, we carry out a backwards exploration from the start vertex to determine those vertices connectable to the start vertex by a path containing none of the vertices on the current simple path. We then extend the current simple path by adding one of the connectable vertices, avoiding possible extensions which don't give rise to cycles. If there is only one way to extend the path,we extend the path in this unique way, avoiding any more backward searching until we are presented with a choice of two alternatives. The complete algorithm is presented below in an Algol-like notation. It uses two list structure representations of the problem graph G: for any vertex v, $A(v)$ is the set of vertices w such that (v,w) is an edge of G, and $B(v)$ is the set of vertices u such that (u,v) is an edge of G.

```
procedure CYCLE; begin comment s is current cycle start vertex;
   procedure BACKTRACK (v); begin comment vertex v is end of current path;
      declare C a set variable local to procedure BACKTRACK;
      mark all vertices unscanned and unconnectable;
      mark s connectable;
    C1: while some vertex $x \geq s$ is connectable and unscanned do begin
         for $u \in B(x)$ and $u \geq s$ do
            if u not on current path then mark u unconnectable;
         mark x scanned;
      end;
      let $C = \{v \mid v \in A(v)$ and v is connectable$\}$;
    C2: for $w \in C$ do begin
         mark all vertices unscanned and unconnectable;
         mark s connectable
      C3:  while some vertex $x \geq s$ is connectable and unscanned do begin
            for $u \in B(x)$ and $u \geq s$ do
            if u is not on current path then mark u connectable;
            mark x scanned;
         end;
         add w to current path;
      C4:  while $w \neq s$ and A(w) contains exactly one connectable vertex x do begin
            w:=x;
            add w to current path;
         end;
         if $w = s$ then output cycle else BACKTRACK (w);
         delete vertices after v from current path;
      end;
   end;
```

```
C5: do a depth-first search of problem graph. Divide graph into strongly
      connected components.  Locate vertices with entering cycle arcs.
      Mark these as cycle start vertices;
    for each strongly connected component do
      for each cycle start vertex s do begin
        C6:  BACKTRACK (s);
      end;
end;
```

In this algorithm, Step C5 may be implemented to run in O(V+E) time as described in [10]. Steps C1 and C3 are backward searches which require O(E) time. Because of the checks in steps C1, C3, and C5, the set C will be non-empty when calculated during any call of BACKTRACK. Furthermore, because of Step C4, in any nested call of BACKTRACK the set C will contain more than one element. (The nested calls on BACKTRACK are those not initiated in Step C6.). Each nested call on BACKTRACK thus gives rise either to two or more simple cycles or to two or more new nested calls on BACKTRACK. Therefore the total number of calls on BACKTRACK is O(C) where C is the number of simple cycles. Suppose we charge the O(E) time spent outside the while loop C2 during a given call on BACKTRACK to that call, and we charge the O(E) time spent during one execution of while loop C2 to the resultant simple cycle or nested call on BACKTRACK. Then the total time required by algorithm CYCLE is O(V+E+EC). CYCLE requires O(V+E) storage plus space for the output cycles. If the simple cycles are used as they are generated, CYCLE requires O(V+E) total storage space.

CYCLE may easily be modified so that it lists all cycles (no duplicate edges) or all simple paths from some set of start vertices to a set of finish vertices, or all paths (no duplicate edges) from a set of start vertices to a set of finish vertices. The time bounds for these variations of the algorithm are the same as that for the original algorithm: $O(V+E+EN)$, where N is the number of objects to be listed.

Conclusions:

This paper has presented algorithms for listing spanning trees, cycles, and other kinds of paths in graphs. The algorithms all have $O(V+E+EN)$ time bounds, where N is the number of objects to be listed. The algorithms are both theoretically efficient and easy to program. They are based on restricted backtracking.

Several other less-understood problems may be susceptible to the type of analysis performed here. Perhaps the most interesting of these is the problem of enumerating all the cliques of an undirected graph. Many algorithms appear in the literature (see [29]); the best theoretically seems to be Bierstone's [3,30] which is not a backtracking algorithm but which has a worst-case time bound proportional to N^2 if the problem graph contains N cliques. This is the only algorithm for which a worst-case time bound polynomial in the number of cliques has been proved. Since the number of cliques may be quite large, this algorithm may be inefficient. Experimental evidence [29] suggests that a recursive algorithm due to Bron and Kerbosch is generally faster, even though no good theoretical bound on its running time is known. It is an open question whether there is a clique listing algorithm with a worst-case time bound linear in the number of cliques.

REFERENCES

[1] MacWilliams, F.J. "Topological network analysis as a computer program", IRE Trans.,Vol.CT-5 (1958), pp. 228-229.

[2] Allen, F.E. "Program optimization", Annual Review in Automatic Programming, Vol. 5, Pergammon Press, 1969, pp. 239-307.

[3] Augustin, J.G., and Minker, J. "An analysis of some graph theoretical cluster techniques", J. ACM, Vol. 17, No. 4, (October 1970), pp. 571-588.

[4] Floyd, R.W. "Nondeterministic algorithms", J.ACM, Vol. 14, No. 4, (October 1967), pp. 636-644.

[5] McIlroy, M.D. "Generation of spanning trees (Algorithm 354)." Comm. ACM, Vol. 12, No. 9 (September 1969), p. 511.

[6] Minty, G.J. "A simple algorithm for listing all the trees of a graph", IEEE Trans. on Circuit Theory, Vol. CT-12 (1965), p. 120.

[7] Feussner, W. "Uber Stromberzeigung in netzformigen Leitern", Annalen der Physik, Vol. 9, (1902), pp. 1304-1329 also Ibid. Vol. 15 (1904), pp. 385-394.

[8] Hakimi, S.L. and Green, D.G. "Generation and realization of trees and k-trees", IEEE Trans.on Circuit Theory, Vol. CT-11, (1964), pp. 247-255.

[9] Watanabe, H. "A computational method for network topology", IRE Trans. Vol. CT-7, (1960), pp. 296-302.

[10] Tarjan, R. "Depth-first search and linear graph algorithms", SIAM J. Comput. Vol. 1, (1972), pp. 146-160.

[11] Hopcroft, J.E. and Tarjan, R. "Efficient algorithms for graph manipulation (Algorithm 447)", Comm. ACM, Vol. 16, No. 6, (June 1973), pp. 372-378.

[12] Tarjan, R. "A note on finding the bridges of a graph", Information Processing Letters, to appear.

[13] Read, R.C., "Branching techniques in the application of computers to graph theory problems", Department of Combinatorics and Optimization, University of Waterloo, unpublished manuscript.

[14] Fischer, M.J. "Efficiency of equivalence algorithms", Complexity of Computer Computations, R.E. Miller and J.W. Thatcher (eds.), Plenum Press, New York, 1972, pp. 157-168.

[15] Hopcroft, J.E. and Ullman, J.D. "Set-merging algorithms", SIAM J. Comput., Vol. 2, No. 4, (December 1973), pp. 294-303.

[16] Tarjan, R. "Efficiency of a good but not linear set union algorithm," submitted to J.ACM.

[17] Tiernan, J.C. "An efficient search algorithm to find the elementary cycles of a graph," Comm. ACM, Vol. 13, (1970) pp. 722-726.

[18] Weinblatt,A. "A new search algorithm for finding the simple cycles of a finite directed graph," J. ACM, Vol. 19 (1972), pp. 43-56.

[19] Tarjan, R. "Enumeration of the elementary circuits of a directed graph," SIAM J. Comput., Vol. 2, No. 3 (September 1973) pp. 211-216 .

[20] Sloane, N.J.A. "On finding the paths through a network," Bell System Technical Journal, Vol. 51, No. 2 (February 1972) pp. 371-390.

[21] Ehrenfeucht, A., Fosdick, L.D., and Osterweil, L.J. "An algorithm for finding the elementary circuits of a directed graph," Report No. CU-CS-024-73, Department of Computer Science, University of Colorado, August 1973.

[22] Hsu, H.T., and Honkanen, P.A. "A fast minimal storage algorithm for determining all the elementary circuits of an undirected graph," Seventh Annual Princeton Conference on Information Sciences and Systems, March 1973, pp. 419-423.

[23] Johnson, D.B. "Finding all the elementary circuits of a directed graph," Tech. Report No. 145, Computer Science Department, Pennsylvania State University, November 1973.

[24] Prabhaker M. and Deo, N. "On algorithms for finding all circuits of a graph," Tech. Report UIUCDCS-R-73-585, Department of Computer Science, University of Illinois at Urbana-Champaigne, June 1973.

[25] Welch, J.T. "A mechanical analysis of the cyclic structure of undirected linear graphs," J. ACM, Vol. 13, (1966) pp. 205-210.

[26] Yan, S. "Generation of all Hamiltonian circuits, paths, and centers of a graph, and related problems." IEEE Trans. on Circuit Theory, CT-14 (1967) pp. 79-81.

[27] Berztiss, A.T. "A k-tree algorithm for simple cycles of a directed graph," Tech. Report 73-6, Dept. of Computer Science, University of Pittsburgh, May 1973.

[28] Tarjan, R. "Finding dominators in directed graphs," SIAM J. Comput. to appear.

[29] Mulligan, G.D. "Algorithms for finding cliques of a graph," Tech. Report No. 41, Dept. of Computer Science,University of Toronto, May 1972.

[30] Bierstone, E. "Cliques and generalized cliques in a finite linear graph," University of Toronto, October 1967, unpublished report.

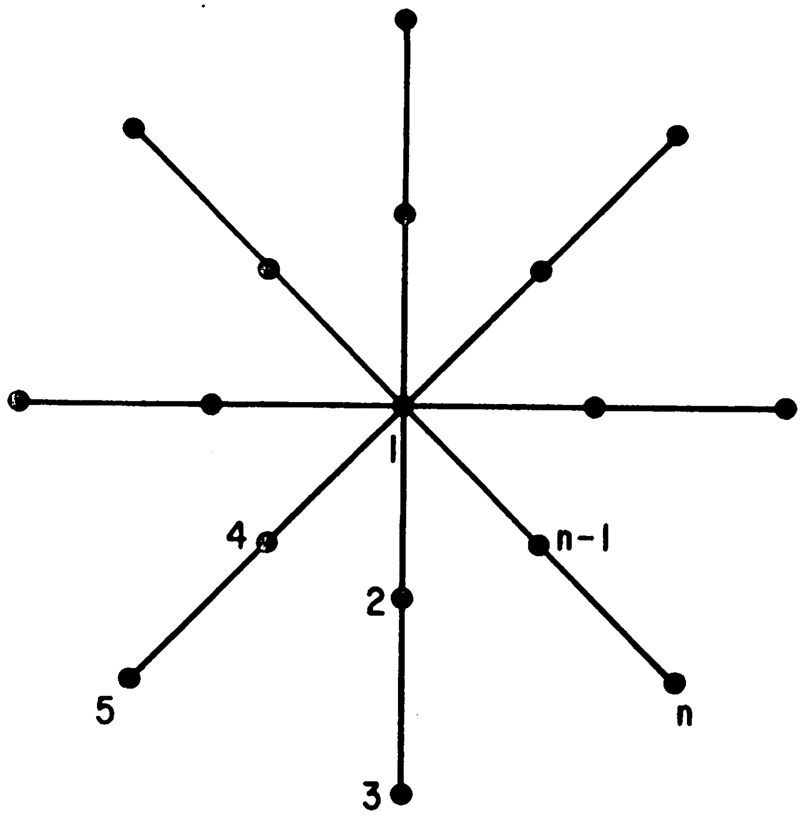

Figure 1: An example on which McIlroy's algorithm is inefficient.