3. Rubin, I., "Message Path Delays in Packet-Switching Communication Networks," *IEEE Trans. on Communications*, Vol. COM-23, No. 2, February 1975, pp. 186-192.

4. Rubin, I., "Tandem Queues with Constant Channel Service Times and Group Arrivals," *Technical Report UCLA-ENG-7417*, University of California, Los Angeles, March 1974.

5. Kleinrock, L., *Communication Nets: Stochastic Message Flow and Delay*, McGraw-Hill, New York, 1964.

6. Frank, H. and I. T. Frisch, "Planning Computer-Communication Networks," *Computer-Communication Networks*, ed. by N. Abramson and F. F. Kuo, Prentice-Hall, Englewood Cliffs, New Jersey, 1973.

7. Ford, L. R. and D. R. Fulkerson, *Flows in Networks*, Princeton University Press, Princeton, New Jersey, 1962.

8. Frank, H. and I. T. Frisch, *Communication, Transmission, and Transportation Networks*, Addison-Wesley, Reading, Massachusetts, 1971.

9. Hakimi, S. L. and S. S. Yau, "Distance Matrix of a Graph and its Reliability," *Quart. Appl. Math.*, Vol. 22, No. 4, 1965, pp. 305-317.

10. Frank, H. and W. Chou, "Routing in Computer Networks," *Networks*, Vol. 1, No. 2, 1971, pp. 99-112.

11. Fratta, L., M. Gerla and L. Kleinrock, "The Flow Deviation Method: An Approach to Store-and-Forward Communication Network Design," *Networks*, Vol. 3, No. 2, 1973, pp. 97-133.

This work was supported by the Office of Naval Research under Grant Number N00014-69-A-0400-4041.

Paper received May 2, 1974.

(1975)

Bounds on Backtrack Algorithms for Listing Cycles, Paths, and Spanning Trees

R. C. Read
University of Waterloo
Waterloo, Ontario, Canada

R. E. Tarjan
University of California
Berkeley, California

ABSTRACT

Backtrack algorithms for listing certain kinds of subgraphs of a graph are described and analyzed. Included are algorithms for listing all spanning trees, all cycles, all simple cycles, or all of certain 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.

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 [15], studying program flow by using the cycles in a program flow graph [1], and organizing information according to the cliques of an associated data graph [2]. 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 *backtracking* [8].

(1975)

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 backtracking process, exploring only fruitful possibilities, and thus list the desired subgraphs much more efficiently than if we do not restrict 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* $G = (V, E)$ is a collection of *vertices* V and *edges* 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. If G is undirected, a cycle $(v_1,v_2), (v_2,v_3), \ldots, (v_{n-1},v_1)$ and its reversal $(v_1,v_{n-1}), (v_{n-1},v_{n-2}), \ldots, (v_2,v_1)$ are also regarded as being the same cycle. A graph $G' = (V',E')$ is a *subgraph* of G if $V' \subseteq V$ and $E' \subseteq E$. 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 verti of G. A *complete graph* is a graph with any two distinct verti joined by an edge. A *clique* of a graph G is a maximal complet subgraph of G. A *directed, rooted* tree is a directed graph wi 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 undirecte graph has an application in the solution of linear electrical networks [15], and several algorithms have been published (e.g [6,9,15,16,17,18,30]). Many of them use backtracking restrict in some way. If T is the number of spanning trees of the grap we would like to have an algorithm which runs in a time bound polynomial in V, E, and T, and which requires as little storag as possible. Suppose we generate all subsets of the edges of a graph by backtracking, and list those which give spanning tree This simple algorithm was apparently first suggested by Feussn [6]. Some of the subsets are spanning trees, but others are n In fact, if the original graph is a tree, such an unrestricted backtracking algorithm will find the single spanning tree very fast but will require at least $k2^E$ time (for some positive constant k) to verify it is the only spanning tree. Thus, we must restrict the backtracking to get an efficient algorithm.

One of the algorithms in the literature, McIlroy's [17], i a minor variation of unrestricted backtracking. McIlroy's algo rithm works vertex by vertex, "growing" a spanning tree by addi one vertex at a time to it. At a given step the algorithm pick a vertex adjacent to the current tree and connects it to the tr using some edge. When the algorithm backtracks to this vertex, it deletes the previous connecting edge from the graph and pick another connecting edge. When no more possible connecting edge remain to be chosen, the algorithm picks another vertex adjacen 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. However, consider the graph in Figure 1. Suppose vertex 1 is examined first. It is not difficult to see th McIlroy's algorithm will find the single spanning tree very fas but will then spend an amount of time exponential in E fruitlessly searching for other spanning trees. Thus it is not an efficient algorithm in the worst case.

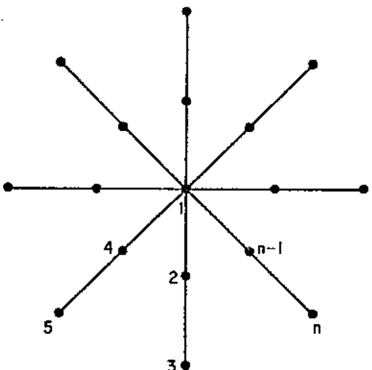

Fig. 1 An example on which McIlroy's algorithm is inefficient.

However, there are algorithms in the literature which *are* efficient. One, devised by Mayeda and Seshu [16], can be implemented to run in O(V+E+VET) time and O(VE) space, if T is the number of trees to be listed. (These are our bounds; Mayeda and Seshu provide no complexity analysis for their algorithm.) An even better algorithm was described informally by Minty [18]; 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 recursive algorithm for listing spanning trees.

```
procedure SPAN; begin
    procedure REC; begin
        declare e a variable local to procedure REC;
        declare B a set variable local to procedure REC;
        comment B should be stored as a linked list;
        if all edges in graph are in current partial spanning
                tree then output spanning tree else begin
            let e be an edge not in partial tree;
            add e to partial tree;
        S1: let B be the set of edges not in partial tree
                joining vertices already connected in par-
                tial tree;
            delete all edges in B from graph;
            REC;
            add all edges in B to graph;
            delete e from partial tree and from graph;
        S2: let B be the set of bridges which are not tree
                edges;
            add all edges in B to partial tree;
            REC;
            remove all edges in B from partial tree;
            add e to graph;
        end;
    end REC;
S3: if graph not connected then output no trees
            else begin
    S4: initialize current partial spanning tree to
            contain all bridges of original graph;
        REC;
    end;
end SPAN;
```

This algorithm is very easy to program. Step S1 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 distinguish ing numbers, and finally adding to L each edge which joins two vertices having the same number. The total time for these ope ations is O(V+E). Steps S2 and S4 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 [11,24,27]. The connectedness test in 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 spa ning trees, and algorithm SPAN will indicate this in O(V+E) ti 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 plu (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 spanni trees. It follows that the number of calls on REC is O(T), wh T is the number of spanning trees. The total running time of SPAN is thus O(V+E+ET).

Consider the storage requirements of SPAN. Any edge in B at some level of the recursion is either deleted from the grap (if it forms a cycle with edges in the partial tree) or in the partial tree (if it is a bridge) at the next-inner level of re cursion. Thus such an edge cannot be in B in any inner recurs level, and the sets B in the various levels of recursion are a ways pairwise disjoint.

Hence the total storage requirements for B over all levels of recursion are O(E). Since the depth of recursion is bounded by E and the original graph requires O(V+E) storage if it is represented by adjacency lists, the total storage required by SPAN is O(V+E) plus storage space for the generated spanning trees (if they are not used as they are generated).

Minor modifications can improve the time and space requirements of SPAN by a constant factor. For instance, various steps of SPAN, such as Step S4 and the connectivity test in S3, can be combined. However, 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 positive 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 [20] coupled with a good algorithm for computing equivalence relations [7,12,28], 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 $\quad t(t+1) = (\sum_{i=1}^{k} (n_i-1))^2 + \sum_{i=1}^{k} (n_i-1)$

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

Thus,

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

By combining each subset of the edges of *J'* with an appropriate subset of the edges of *J*, we may form 2^t different spanning trees of G.

Q.E.D.

Corollary 1: SPAN's worst case running time is within a factor of k max(1, log T log log T) of best possible, for some positive constant k.

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

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

It follows from the proof of Theorem 1 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 $\log\log T \geq \frac{1}{2}\log(E-V+1) \geq \frac{1}{2}\log V$. Since $T \leq V^{V-2}$ [10]; $\log T \leq V \log V$. Combining, we have

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

Q.E.D

Theorem 2: Given any V and E, there is a connected graph with V vertices, E edges, and fewer than t^{t-2} *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-2} spanning trees (there are t^{t-2} spanning trees in a complete graph on t vertices [10]). 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.

Q.E.D.

Theorem 2 shows that the bound in Theorem 1 is reasonably tight.

LISTING SIMPLE CYCLES

Backtracking is also very useful for listing the simple :ycles (and certain other kinds of paths) in an undirected or a lirected graph. A list of simple cycles is useful in optimizing :omputer programs, and several published papers contain algo- :ithms (e.g., [3,5,13,14,20,22,23,25,29,31,32,33]). The two ıost common approaches are to use some kind of restricted back- :racking or to represent cycles as elements in a vector space 'hose basis is a set of fundamental cycles. Prabhaker and Deo 20] discuss these and other methods.

The vector space approach has two significant drawbacks. ırst, it is only applicable to undirected graphs. Second, for ach of the algorithms in [13,32] it is possible to construct a raph with a small number of cycles for which the given algo- ithm requires an exponential amount of time to list the simple ycles. This second drawback (inefficiency) is shared by the lgorithms described in [3,22,29,31,33]. Reference [25] con- ains bad example graphs for the algorithms of Tiernan [29] and einblatt [31]; bad examples for the other algorithms are easily onstructed.

The four reasonably efficient algorithms in the literature ;,14,23,24] all use the same type of backtracking, though re- :ricted in slightly different ways. Suppose we somehow number ıe vertices of the problem graph from 1 to V. Then we can pick ıe smallest numbered vertex on a cycle to be a unique "start" ·rtex of the cycle. For a fixed start vertex, we choose an

edge leading from the vertex. This starts a path in the probl graph. To extend the path, we pick an edge leading from the last vertex on the path. We do not allow the path to intersec itself or to contain vertices smaller than the start vertex. the last vertex on the path has an edge leading to the start v tex, we output a cycle. When we have tried all possible ways extending a path, we back up, delete the last edge on the path and try another possibility. To enumerate all simple cycles, try this backtracking procedure with each vertex as a starting vertex. Tiernan was the originator of this algorithm, which i sometimes very inefficient because many ways of extending a pa may not lead to a simple cycle [25].

However, it is possible to put restrictions on the back- tracking and get an efficient algorithm. The algorithms in [5 14,23,25] are all of this type. If the problem graph has C si ple cycles, Ehrenfeucht, Fosdick, and Osterweil's algorithm [5 has an O(VE(C+1)) time bound, Tarjan's [25] has an O(VE(C+1)) time bound, Johnson's [14] has an O(V+E+EC) time bound, and Szwarcfiter and Lauer's [23] has an O(V+E+EC) time bound. All four algorithms have O(V+E) space requirements in addition to storage space for the output. A new algorithm with an O(V+E+E time bound appears below. This algorithm was developed indepe dently of Johnson's and Szwarcfiter and Lauer's algorithms and rivals them in simplicity and theoretical efficiency.

To improve Tiernan's essentially unrestricted backtrackin procedure, we need a way to select possible cycle-starting ver- tices and a way to restrict the backtracking. Suppose we explc the problem graph using depth-first search [24], 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 de- scendants to ancestors in the tree, called *cycle arcs;* a set of edges from ancestors to descendants in the tree, called *forwarc arcs;* and a set of edges joining unrelated vertices in the spar ning 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 (the latter fact is a non-trivial consequence of the depth-first nature of the search [29]). Each cycle arc is an edge of at least one simple cycle, and each cycle must en in a cycle arc. This follows from results in [24,26]. Repeate applications of depth-first search may be used to partition the graph into a set of trees, which together span all the vertices and a set of non-tree arcs. This process requires O(V+E) time [24]. Then the set of cycle-starting vertices is exactly the

set of vertices with entering cycle arcs. 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 recursive backtracking procedure to build up a simple path which may be extended into a simple cycle. Suppose v is the last vertex on the current path p. We use a search to determine a new extension vertex w such that (v,w) is an edge and there is a path from w to s which avoids p (except at s). For each such extension vertex, we conduct a search to determine how far p plus w can be extended uniquely toward s. We extend the path until two choices for the next vertex are possible, and then we call the backtracking procedure recursively. This extension (without recursion), until two choices are possible, is what makes the algorithm efficient, just as it is what makes the spanning tree algorithm efficient.

The complete algorithm is presented below in an Algol-like notation. It uses a set of adjacency lists to represent the graph G: for any vertex v, A(v) is a list of vertices w such that (v,w) is an edge of G. BACKTRACK is the recursive path-extension procedure. For any vertex x, p(x) is a Boolean variable which is set true if x is on the current path.

If the current path p ends in v, w is an extension vertex, and $(w=w_1,w_2), (w_2,w_3), \ldots, (w_{n-1},w_n=s)$ is a way found to extend p plus w into a simple cycle, recursive procedure DFS is used to determine the first w_i from which there are two or more path extensions which lead to s. DFS is a depth-first search. All vertices x explored by DFS, as well as vertices on the current path, have d(x) set true. The variable flag becomes true as soon as an alternate route to s is found.

```
procedure CYCLE; begin comment s is current cycle start vertex;
    procedure BACKTRACK(v); begin comment vertex v is end of
            current path;
        declare z a variable local to procedure BACKTRACK;
        procedure DFS(u); begin comment this procedure ex-
              plores graph to find an alternate route to s;
              d(u) := true;
              for x ε A(u) do while ¬ flag do
                  if x = s then flag := true else if ¬ d(x)
                      then DFS(x);
        end DFS;
        z :=0;
C1:     while there exists a vertex w ε A(v) with w > z such
                  that there is a path from w to s which
                  avoids the current path do begin
C2:         let (w=w_1,w_2), (w_2,w_3), ..., (w_n-1,w_n=s) be a
                  path from w to s avoiding vertices (except
                  s) on current path;
            z := w;
            i := 1;
            flag := false;
            for all vertices x do d(x) := p(x);
            comment add vertices w_1, w_2, ..., to current pat
                  until a choice of ways to reach s is found;
C3:         while (w_i≠s) and ¬ flag do begin
                  add w_i to current path;
                  p(w_i) := d(w_i) := true;
                  for u ε A(w_i) do if (u≠w_i+1) and (d(u) =
                       false) then DFS(u);
                  i := i + 1;
            end;
            if ¬ flag then begin
                  add w_i = s to current path;
                  output cycle;
            end else BACKTRACK(w_i-1);
            delete vertices after v from current path and
                  restore their p values to false;
        end; end BACKTRACK
C4:     do a depth-first search of problem graph. Divide graph
              into strongly connected components, deleting edges
              between components. Locate vertices with entering
              cycle arcs. Mark these as cycle start vertices;
        for v ε V do p(v) := false;
        for each strongly connected component do
              for each cycle start vertex s do begin
              C5:  delete vertices numbered less than s from
                   adjacency lists;
                   p(s) := true;
                   BACKTRACK(s);
                   p(s) := false;
end end CYCLE:
```

After a path $w = w_1, w_2, \ldots, w_n = s$ from w to s is found in Steps C1 and C2, Step C3 adds the vertices on this path one-at-a-time to the current path. Before the next vertex is added, DFS is called to discover if some alternate extension is possible. The non-recursive extension continues until a choice is possible (BACKTRACK is then called recursively) or s is reached (a cycle is generated).

In this procedure, Step C4 may be implemented to run in O(V+E) time as described in [24,26]. In each strongly connected component, the number of edges is at least as great as the number of vertices. Thus the test in C1 for an appropriate w, implemented as a search, requires O(E) time. The path required in Step C2 can be constructed during this search. Since DFS(u) sets d(u) to true, the total time spent in DFS during one execution of Step C3 is O(E), and the total time for one execution of Step C3 is O(E). The total time spent in Step C5 (for all start vertices) is O(E) if pointers are kept to all occurrences of each vertex in the adjacency lists.

Because of the calculations in Steps C1, C2, and C4, at least one extension vertex will be found in C1 during any call of BACKTRACK. Furthermore, because of Step C3, at least two extension vertices will be found in any nested call of BACKTRACK. (The nested calls on BACKTRACK are those initiated in Step C3.) Each extension vertex leads to an execution of loop C1 and thus either generates a cycle or generates a new call on BACKTRACK. Thus each nested call on BACKTRACK gives rise either to two or more simple cycles, to two or more nested calls on BACKTRACK, or to at least one simple cycle and one nested call. Therefore, the total number of calls on BACKTRACK is O(C) where C is the number of simple cycles.

A given call on BACKTRACK requires O(E) time spent outside loop C1, O(E) time per complete execution of loop C1, O(E) time for the final (failing) test of the condition in C1, and time for one nested call on BACKTRACK per complete execution of C1. Suppose we charge the O(E) time spent outside loop C1 and in the final test of the C1 condition during a given call on BACKTRACK to that call, and we charge the O(E) time spent during one complete execution of C1 to the resultant simple cycle or nested call on BACKTRACK. Then the total time required by algorithm CYCLE is O(V+E+EC).

A call on BACKTRACK requires only a finite amount of storage. The maximum possible depth of recursion is V and the storage required by other variables is O(V+E), so the total storage required by CYCLE is O(V+E), plus space for the output cycles if they are not used as they are generated.

CYCLE may easily be modified so that it lists all cycles (no duplicate edges). This is accomplished by encoding the cu rent path as a list of edges rather than a list of vertices a not allowing an edge to appear twice in the current path. CY may also be modified to list all simple paths from some set S of start vertices to a set F of finish vertices. In this cas a backward search is made from the finish vertices to elimina useless start vertices and BACKTRACK(s) is called for each st vertex s which will give a path. The tests "$x = s$," "$w_i = s$" inside BACKTRACK need to be replaced by tests "$x \in F$," "$w_i \in$ F" Similarly, CYCLE may be modified to list all paths (no duplic edges) from a set of start vertices to a set of finish vertic Other variations may suggest themselves. The time bounds for these variations of the algorithm are the same as that for th original algorithm: O(V+E+EN), where N is the number of objec to be listed. The space bounds for the variations are O(V+E) plus space for the output if required.

CONCLUSIONS

This paper has presented backtrack algorithms for listin spanning trees, cycles, and other kinds of paths in graphs. algorithms all have O(V+E+EN) time bounds, where N is the num of objects to be listed. The algorithms are both theoretical efficient and easy to program. They are based on two ideas: (1) restricting backtracking so that only fruitful possibilit are explored and (2) avoiding recursion until at least two fr ful choices are possible.

Several other less-understood problems may be susceptibl to the type of analysis performed here. Perhaps the most int esting of these is the problem of listing all the cliques of a undirected graph. Many algorithms appear in the literature (s [19]); the best theoretically seems to be Bierstone's [2,4] wh is not a backtracking algorithm but which has a worst-case ti bound proportional to N^2 if the problem graph contains N cliq This is the only algorithm for which a worst-case time bound polynomial in the number of cliques has been proved. Since th number of cliques may be quite large, this algorithm may be in efficient. Experimental evidence [19] suggests that a recurs algorithm due to Bron and Kerbosch is generally faster, even though no good theoretical bound on its running time is known.

Note added in proof: S. Tsukiyama has recently devised a clique-finding algorithm with running time linear in the number of cliques (I. Shirikawa, private communication).

REFERENCES

1. Allen, F. E., "Program Optimization," *Annual Review in Automatic Programming*, Vol. 4, Pergammon Press, 1969, pp. 239-307.

2. Augustin, J. G. and J. Minker, "An Analysis of Some Graph Theoretical Cluster Techniques," *J. ACM*, Vol. 17, No. 4, October 1970, pp. 571-588.

3. Berztiss, A. T., "A k-Tree Algorithm for Simple Cycles of a Directed Graph," *Tech. Report 73-6*, Department of Computer Science, University of Pittsburg, May 1973.

4. Bierstone, E., "Cliques and Generalized Cliques in a Finite Linear Graph," unpublished report, University of Toronto, October 1967.

5. Ehrenfeucht, A., L. D. Fosdick and L. J. Osterweil, "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.

6. Feussner, W., "Uber Stromberzeigung in Netzformigen Leitern," *Annalen der Physik*, Vol. 9, 1902, pp. 1304-1329, also Ibid. Vol. 15, 1904, pp. 385-394.

7. 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.

8. Floyd, R. W., "Nondeterministic Algorithms," *J. ACM*, Vol. 14, No. 4, October 1967, pp. 636-644.

9. Hakimi, S. L. and D. G. Green, "Generation and Realization of Trees and k-Trees," *IEEE Trans. on Circuit Theory*, Vol. CT-11, 1964, pp. 247-255.

10. Harary, F., *Graph Theory*, Addison-Wesley, Reading, Massachusetts, 1969, pp. 152-154.

11. Hopcroft, J. E. and R. Tarjan, "Efficient Algorithms for Graph Manipulation (Algorithm 447)," *Comm. ACM*, Vol. 16, No. 6, June 1973, pp. 372-378.

12. Hopcroft, J. E. and J. D. Ullman, "Set-Merging Algorithms *SIAM J. COMPUT.*, Vol. 2, No. 4, December 1973, pp. 294-30

13. Hsu, H. T. and P. A. Honkanen, "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.

14. Johnson, D. B., "Finding All the Elementary Circuits of a Directed Graph," *Tech. Report No. 145*, Computer Science Department, Pennsylvania State University, November 1973.

15. MacWilliams, F. J., "Topological Network Analysis as a Computer Program," *IRE Trans.*, Vol. CT-5, 1958, pp. 228-229.

16. Mayeda, W. and S. Seshu, "Generation of Trees Without Duplications," *IEEE Trans. on Circuit Theory*, Vol. CT-12 1965, pp. 181-185.

17. McIlroy, M. D., "Generation of Spanning Trees (Algorithm 354)," *Comm. ACM*, Vol. 12, No. 9, September 1969, p. 511

18. Minty, G. J., "A Simple Algorithm for Listing All the Tr of a Graph," *IEEE Trans. on Circuit Theory*, Vol. CT-12, 1965, p. 120.

19. Mulligan, G. D., "Algorithms for Finding Cliques of a Gr *Tech. Report No. 41*, Department of Computer Science, University of Toronto, May 1972.

20. Prabhaker, M. and N. Deo, "On Algorithms for Finding All Circuits of a Graph," *Tech. Report UIUCDCS-R-73-585*, Department of Computer Science, University of Illinois at Urbana-Champaign, June 1973.

21. Read, R. C., "Branching Techniques in the Application of Computers to Graph Theory Problems," *Research Report COR 74-10*, Department of Combinatorics and Optimization, University of Waterloo, 1974.

22. Sloan, N. J. A., "On Finding the Paths Through a Network *BSTJ*, Vol. 51, No. 2, February 1972, pp. 371-390.

23. Szwarcfiter, J. L. and P. E. Lauer, "Finding the Elementary Cycles of a Directed Graph in O(N + M) Per Cycle," *Tech. Report No. 60*, Computing Laboratory, University of Newcastle Upon Tyne, May 1974.

24. Tarjan, R., "Depth-First Search and Linear Graph Algorithms," *IRE Trans.*, Vol. 1, 1972, pp. 146-160.

25. Tarjan, R., "Enumeration of the Elementary Circuits of a Directed Graph," *SIAM J. Comput.*, Vol. 2, No. 3, September 1973, pp. 211-216.

26. Tarjan, R., "Finding Dominators in Directed Graphs," *SIAM J. Comput.*, Vol. 3, No. 1, March 1974, pp. 62-89.

27. Tarjan, R., "A Note on Finding the Bridges of a Graph," *Information Processing Letters*, Vol. 2, No. 6, April 1974, pp. 160-161.

28. Tarjan, R., "Efficiency of a Good But Not Linear Set Union Algorithm," *J. ACM*, Vol. 22, No. 2, April 1975, pp. 215-225.

29. Tiernan, J. C., "An Efficient Search Algorithm to Find the Elementary Cycles of a Graph," *Comm. ACM*, Vol. 13, 1970, pp. 722-726.

30. Watanabe, H., "A Computational Method for Network Topology," *IRE Trans.*, Vol. CT-7, 1960, pp. 296-302.

31. Weinblatt, A., "A New Search Algorithm for Finding the Simple Cycles of a Finite Directed Graph," *J. ACM*, Vol. 19, 1972, pp. 43-56.

32. Welch, J. T., "A Mechanical Analysis of the Cyclic Structure of Undirected Linear Graphs," *J. ACM*, Vol. 13, 1966, pp. 205-210.

33. Yau, 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.

Research of first author was partially supported by the National Research Council of Canada, Grant #A8142.

Research of second author was partially supported by the National Science Foundation, Contract No. GJ-35604X, and by a Miller Research Fellowship.

Paper received June 17, 1974.

The Minimum Number of Edges and Vertices in a Graph with Edge Connectivity n and m n-Bonds

R. E. Bixby
University of Kentucky
Lexington, Kentucky

ABSTRACT

The problem studied is the following: What is the minimum number of edges and vertices in a graph with edge connectivity n and exactly m n-bonds (cuts)? It is perhaps surprising that this problem turns out to have an essentially closed form solution for all m and n (Theorem C at the end of Section 5). Furthermore, the methods employed make it possible, for many values of m and n, to actually write down a graph that achieves the minimum. These methods involve, as an interesting by-product, estimating the solutions of integer minimization problems of the form $\min\{r_0+r_1+\ldots+r_k : \binom{r_0}{2}+\binom{r_1}{2}+\ldots+\binom{r_k}{2}=m,\ \min r_i \geq 2\}$ *where k varies as part of the minimization. In particular, it is shown that the obvious "greedy" algorithm (i.e., choose* r_0 *as large as possible, then* r_1 *as large as possible and so forth until m is exhausted) almost works.*

0. DEFINITIONS

The material in this section is, for the most part, reasonably standard. The knowledgeable reader may therefore wish to read only the latter part of the section, beginning with the definition of C(G), and then move on to the Introduction (Section 1).

A (finite) *graph* G consists of a finite set E(G) of *edges* and a finite set V(G) of *vertices*, together with a relation of incidence which associates with each edge two vertices, not necessarily distinct, called its *ends*; an edge is said to join its ends. Note that our definition of graph allows for so-call