SIAM J. COMPUT.
Vol. 14, No. 1, February 1985

ARBORICITY AND SUBGRAPH LISTING ALGORITHMS*

NORISHIGE CHIBA† AND TAKAO NISHIZEKI†

Abstract. In this paper we introduce a new simple strategy into edge-searching of a graph, which is useful to the various subgraph listing problems. Applying the strategy, we obtain the following four algorithms. The first one lists all the triangles in a graph G in $O(a(G)m)$ time, where m is the number of edges of G and $a(G)$ the arboricity of G. The second finds all the quadrangles in $O(a(G)m)$ time. Since $a(G)$ is at most three for a planar graph G, both run in linear time for a planar graph. The third lists all the complete subgraphs K_l of order l in $O(la(G)^{l-2}m)$ time. The fourth lists all the cliques in $O(a(G)m)$ time per clique. All the algorithms require linear space. We also establish an upper bound on $a(G)$ for a graph G: $a(G) \leqq \lceil (2m+n)^{1/2}/2 \rceil$, where n is the number of vertices in G.

Key words. arboricity, clique, complete subgraph, independent set, quadrangle, subgraph listing algorithm, triangle

1. Introduction.

The problems to list certain kinds of subgraphs of a graph arise in many practical applications [2], [3], [4], [6], [8], [10], [11]. In this paper we introduce a new simple strategy into edge-searching of a graph, which is useful to the various subgraph listing problems. We choose a vertex v in a graph and scan the edges of the subgraph induced by the neighbors of v to find the pattern subgraphs containing v. The feature of the strategy is to repeat the searching above for each vertex v in nonincreasing order of degree and to delete v after v is processed so that no duplication occurs. We will show in the succeeding section that the procedure above requires $O(a(G)m)$ time. Throughout this paper m is the number of edges of a graph G, n is the number of vertices of G, and $a(G)$ is the *arboricity* of G, that is, the minimum number of edge-disjoint spanning forests into which G can be decomposed [5]. We use the rather unfamiliar graph invariant $a(G)$ as a parameter in bounding the running time of algorithms.

The strategy yields simple algorithms for the problems to list certain kinds of subgraphs of a graph. The kinds of these subgraphs include "triangle," "quadrangle," "complete subgraph of a fixed order," and "clique." Our algorithms are as fast as the known ones if any, and a factor n is often reduced to $a(G)$ in the time complexity.

In § 2 we give an upper bound on $a(G)$ for a general graph G: $a(G) \leqq \lceil (2m+n)^{1/2}/2 \rceil$, which implies $a(G) \leqq O(m^{1/2})$ for a connected graph G. In § 3 we give a simple algorithm which lists all the triangles in an arbitrary graph G in $O(a(G)m)$ time. In § 4 we present an $O(a(G)m)$ time algorithm for finding all the quadrangles (i.e. C_4) in G, which does not actually list C_4 but finds a representation of all the C_4. If G is planar, $a(G) \leqq 3$, so these two algorithms run in linear time for planar graphs. Because of the bound on $a(G)$, they run in at most $O(m^{3/2})$ time for general graphs. In § 5, extending the triangle listing algorithm, we present an $O(la(G)^{l-2}m)$ time algorithm for listing all the complete subgraphs of order l (i.e. K_l) in G, where l is an arbitrary number. Finally in § 6 we present an algorithm for listing all the cliques in G in $O(a(G)m)$ time per clique. All our algorithms require linear space and exceed the known algorithms [3], [6], [9] for the same purposes in running time, space, or simplicity.

* Received by the editors December 15, 1982.

† Department of Electrical Communications, Faculty of Engineering, Tohoku University, Sendai 980, Japan.

2. Preliminaries. We first define some terms. Let $G=(V, E)$ be a *simple graph* with vertex set V and edge set E. The edge set of graph G is often denoted by $E(G)$. The edge joining vertices u and v is denoted by (u, v). Throughout this paper we denote by n the number of vertices and by m the number of edges of a graph. Let $d(v)$ denote the *degree* of a vertex v, that is, the number of edges incident to v. A graph is *planar* if it is embeddable on the plane without edge crossing. It is well-known that $m \leqq 3n-3$ if G is planar [5]. A *triangle* in a graph is a *cycle* of length three (i.e. C_3), in other words, a *complete subgraph* of three vertices (i.e. K_3). An *independent set* is a set of pairwise nonadjacent vertices in a graph. A *clique* is a maximal complete subgraph in a graph. We denote by $\lceil x \rceil$ the smallest integer not less than x.

We next present two results; the first is concerned with the arboricity of a graph and the other with the time required by scanning edges in a way of our strategy.

LEMMA 1. *Let a graph G have n vertices and m edges. Then*

(1) (a) $a(G) \leqq \lceil (2m+n)^{1/2}/2 \rceil$;
(b) $a(G) \leqq \lceil n/2 \rceil$; *and*
(c) $a(G) \leqq 3$ *if G is planar* [5, p. 124].

Proof. (a) Nash-Williams [7] showed that

$$a(G) = \max_{H \subset G} \lceil q/(p-1) \rceil, \tag{2}$$

where H runs over all nontrivial subgraphs of G, p is the number of vertices and q the number of edges of H. Suppose that the maximum in the right-hand side of (2) is achieved by a subgraph H having p vertices and q edges. Let k be the number of edges of a complete graph with p vertices, that is, $k = p(p-1)/2$. Consider the following two cases.

Case 1. $k \leqq m$.

$$\begin{aligned} a(G) = \lceil q/(p-1) \rceil \leqq \lceil k/(p-1) \rceil &= \lceil p/2 \rceil \\ &= \lceil (2k+p)^{1/2}/2 \rceil \leqq \lceil (2m+n)^{1/2}/2 \rceil. \end{aligned}$$

Case 2. $k \geqq m$.

$$\begin{aligned} a(G) &= \lceil q/(p-1) \rceil \leqq \lceil m/(p-1) \rceil \leqq \lceil \{mk/(p-1)^2\}^{1/2} \rceil \\ &= \lceil \{(m(p-1)+m)/2(p-1)\}^{1/2} \rceil \\ &\leqq \lceil \{m/2 + k/2(p-1)\}^{1/2} \rceil \\ &= \lceil (2m+p)^{1/2}/2 \rceil \\ &\leqq \lceil (2m+n)^{1/2}/2 \rceil. \end{aligned}$$

(b) Immediate from (2).
(c) If G is planar, (2) implies that

$$a(G) \leqq \max_{H \subset G} \lceil (3p-3)/(p-1) \rceil = 3.$$ Q.E.D.

Since $a(K_n) = \lceil n/2 \rceil = \lceil (2m+n)^{1/2}/2 \rceil$ where $m = n(n-1)/2$, there exist an infinite number of graphs attaining the upper bound in (1). In this sense the bound is best possible. It should be noted that $a(G) = O(1)$ for a large class of graphs including (i) planar graphs, (ii) graphs of bounded genus, and (iii) graphs of bounded maximum degree.

LEMMA 2. *If graph $G=(V, E)$ has n vertices and m edges, then*

$$\sum_{(u,v)\in E} \min\{d(u), d(v)\} \leqq 2a(G)m.$$

Proof. Let F_i $(1 \leqq i \leqq a(G))$ be the edge-disjoint spanning forests of G such that $E(G)=\bigcup_{1\leqq i\leqq a(G)} E(F_i)$. Associate each edge of F_i with a vertex of G as follows: choose an arbitrary vertex u of each tree T in forest F_i as the root of T; regard T as a rooted tree with root u in which all the edges are directed from the root to the descendants; and associate each edge e of tree T with the head vertex $h(e)$ of e. Thus, every vertex of F_i, except the roots, is associated with exactly one edge of F_i. Then we have

$$\begin{aligned}\sum_{(u,v)\in E} \min\{d(u), d(v)\} &\leqq \sum_{1\leqq i\leqq a(G)} \sum_{e\in E(F_i)} d(h(e)) \\ &\leqq \sum_{1\leqq i\leqq a(G)} \sum_{v\in V} d(v) \\ &= 2a(G)m. \end{aligned}$$

Q.E.D.

3. Algorithm for listing triangles. The triangle detection problem often arises in many combinatorial problems such as (1) the minimum cycle detection problem [6], (2) the approximate Hamiltonian walk problem in maximal planar graphs [8], and (3) the approximate minimum vertex cover (or maximum independent set) problem in planar graphs [3], [4]. Itai and Rodeh [6] presented an algorithm for finding all the triangles, which uses an adjacency matrix, so requires $O(n^2)$ space but runs in $O(m^{3/2})$ time for general graphs and in $O(n)$ time for planar graphs. Bar-Yehuda and Even [3] improved the space complexity of the algorithm from $O(n^2)$ into $O(n)$ by avoiding the use of the adjacency matrix. On the other hand Papadimitriou and Yannakakis [9] gave a linear, but a little complicated, algorithm for finding all the complete subgraphs, i.e. K_i $(1\leqq i\leqq 4)$, in a planar graph with assuming a plane embedding of the graph.

Our algorithm for listing triangles in a graph G is very simple as shown below. Observe that each triangle containing a vertex v corresponds to an edge joining two neighbors of v.

```
procedure K3($G$);
  {Let $G$ be a graph with $n$ vertices and $m$ edges.}
  begin
    sort the vertices $v_1, v_2, \cdots, v_n$ of $G$ in such a way that $d(v_1) \geqq d(v_2) \geqq \cdots \geqq$
    $d(v_n)$;
    for $i := 1$ to $n-2$
     do begin
        {find all the triangles containing vertex $v_i$, each of which corresponds
        to an edge joining two neighbors of $v_i$.}
1:      mark all the vertices adjacent to $v_i$;
        for each marked vertex $u$
         do begin
2:           for each vertex $w$ adjacent to $u$
              do if $w$ is marked
                  then print out triangle $(v_i, u, w)$;
3:            erase the mark from $u$
           end;
        {delete $v_i$ from $G$ so that no duplication occurs.}
```

```
4:      delete vertex $v_i$ from $G$ and let $G$ be the resulting graph
    end
end;
```

We have the following result on the algorithm.

THEOREM 1. *Let G be a connected graph with n vertices and m edges. Algorithm K3 lists all the triangles in G in $O(a(G)m)$ time, and especially in $O(n)$ time if G is planar.*

Proof. Since one can easily verify the correctness, we shall show that the algorithm runs in $O(a(G)m)$ time.

Clearly the degrees of vertices can be computed in $O(m)$ time. Since the degree of any vertex is at most $n-1$, one can sort the vertices in $O(n)$ time by the bucket sort [1]. We use doubly linked adjacency lists as a data structure to represent a graph G. The two copies of each edge (u, v), one in the list of v and the other in the list of u, are also doubly linked. Using such a data structure, we can delete a vertex v from G in $O(d(v))$ time, and scan all the vertices adjacent to a vertex v in $O(d(v))$ time. Now consider the time required by the ith iteration of the outmost **for** statement. Statements 1, 3 and 4 require $O(d(v_i))$ time. Statement 2 requires at most $O(\sum_{u \in N(v_i)} d(u))$ time, where $d(u)$ denotes the degree of vertex u in the original graph and $N(v_i)$ denotes the set of neighbors of v_i in the current graph. Therefore the total running time T of the algorithm is bounded as follows:

$$T \leqq O(m)+O(n)+\sum_{v_i \in V} O(d(v_i)+\sum_{u \in N(v_i)} d(u)).$$

Since v_i has the largest $d(v_i)$ among all the vertices in the current graph, we have $d(u) \leqq d(v_i)$ for each $u \in N(v_i)$. Since v_i is deleted at Statement 4, each edge of G is involved exactly once in the double summations above. Thus we have

$$T \leqq O(m)+O(n)+O\left(\sum_{(u,v) \in E} \min\{d(u), d(v)\}\right).$$

Using Lemma 2, we have $T \leqq O(a(G)m)$.

If G is planar, the algorithm runs in $O(a(G)m) \leqq O(n)$ time since $a(G) \leqq 3$ by Lemma 1(c). Q.E.D.

Algorithm K3 is conceptually very simple and easy to implement. Furthermore it is at least as fast as the known ones [3], [6], [9] since $O(a(G)m) \leqq O(m^{3/2})$ by Lemma 1(a).

The benefit of our strategy may be intuitively explained as follows: since we delete the vertices one by one in the largest degree order, the graph tends to become sparse soon; this also prevents the edges incident to a vertex of large degree from being scanned many often.

Applying the strategy, we will give three more algorithms for other subgraphs listing problems in the succeeding sections.

4. Algorithm for finding quadrangles.

In this section, using our searching strategy, we design an efficient algorithm for finding all the quadrangles.

If vertices $u_1, u_2, \cdots, u_l$ ($l \geqq 2$) are all adjacent to two common vertices v and w, that is, these $l+2$ vertices induce a complete bipartite graph $K_{2,l}$, then any quadruple (v, u_i, w, u_j), $1 \leqq i < j \leqq l$, forms a quadrangle. Thus even in a planar graph, there may exist $O(n^2)$ quadrangles. Instead of listing these quadrangles individually, we list a triple $(v, w, \{u_1, u_2, \cdots, u_l\})$ representing them altogether.

Our algorithm C4 depicted below proceeds, for each vertex v of a graph, to find all the quadrangles containing v: for each vertex w within distance two from v, the algorithm finds all such $u_1, u_2, \cdots, u_l$ which are adjacent to both v and w, and stores them in a set $U[w]$. When the quadrangles containing v have been found, v is deleted in order to avoid the duplication.

```
procedure C4($G$);
  {Let $G=(V, E)$ be a graph with $n$ vertices.}
  begin
    sort the vertices in $V$ in a way that $d(v_1) \geqq d(v_2) \geqq \cdots \geqq d(v_n)$;
    for each vertex $v \in V$ do $U[v] := \emptyset$;
    for $i := 1$ to $n$
      do begin
        for each vertex $u$ adjacent to $v_i$
          do for each vertex $w \neq v_i$ adjacent to $u$
            do begin
              $U[w] := U[w] \cup \{u\}$
            end;
        for each vertex $w$ with $|U[w]| \geqq 2$
          do print out the triple $(v_i, w, U[w])$;
        for each vertex $w$ with $U[w] \neq \emptyset$ do $U[w] := \emptyset$;
        delete the vertex $v_i$ from $G$ and let $G$ be the new graph
      end
  end;
```

The graph depicted in Fig. 1 contains seven quadrangles. Algorithm C4 lists the following five triples: $(1, 5, \{2, 7, 10\})$, $(1, 4, \{2, 3\})$, $(3, 8, \{4, 6\})$, $(3, 9, \{4, 6\})$, and $(4, 6, \{8, 9\})$. The first triple represents three quadrangles.

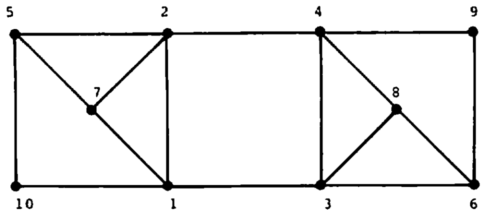

FIG. 1. *A graph containing seven quadrangles.*

We easily obtain the following theorem.

THEOREM 2. *Algorithm C4 obtains a representation of all the quadrangles in a connected graph G in $O(a(G)m)$ time, using $O(m)$ space.*

Note that Algorithm C4 does not store the triples. Since Algorithm C4 runs in $O(a(G)m)$ time, clearly all the quadrangles, if desired, could be represented by the triples in $O(a(G)m)$ space.

5. Algorithm for listing complete subgraphs. Observe the following fact: Algorithm K3 finds a triangle (K_3) containing a vertex v by detecting an edge (K_2) in a subgraph induced by the neighbors of v. In a similar manner, one can find a complete subgraph K_l containing a vertex v by detecting a complete subgraph K_{l-1} in a subgraph induced by the neighbors of v. We first present, for the sake of understanding, a simple recursive algorithm for listing the complete subgraphs K_l of fixed order $l(\geqq 2)$ in a graph $G=(V, E)$.

```
procedure COMPLETE($l$, $G$)
  procedure K($k$, $G_k$);
   {find all $K_k$ in a subgraph $G_k$. $C$ is a global stack.}
   begin
    if $k=2$
     then for each edge $(x, y)$ of $G_k$
           do print out $\{x, y\} \cup C$
     else for each vertex of $G_k$
           do begin
                let $G_{k-1}$ be the subgraph of $G_k$
                 induced by the neighbors of $v$;
                add $v$ to the top of $C$;
                K($k-1$, $G_{k-1}$);     {find $K_{k-1}$ in $G_{k-1}$, which, together
                                          with $v$, form $K_k$ in $G_k$}
                delete $v$ from the top of $C$;
                $G_k := G_k - v$          {delete $v$ to avoid the duplication}
              end
  end;
  begin
   $C := \emptyset$;
   K($l$, $G$)
  end;
```

In the algorithm above Stack C contains a sequence of vertices which have been known to be pairwise adjacent. When procedure K(k, G_k) is executed (at a recursive call of depth $l-k$), C contains $l-k$ pairwise adjacent vertices, and the subgraph G_k contains all the vertices that are adjacent to every vertex in C. Procedure K(k, G_k) finds all the K_k in G_k, each of which, together with the vertices in C, forms a K_l in G. Noting these facts, one can easily verify the correctness of the algorithm by induction on l. However the direct implementation of COMPLETE does not yield an efficient algorithm because it had to produce and store a sequence of induced subgraphs of G.

In order to avoid the trouble above, we introduce a certain kind of vertex-labeling, by which all the vertices are labeled either "l", "$l-1$", $\cdots$, or "k". The vertices labelled "k" induce the subgraph G_k currently processed. Let U be the vertex set of G_k. We order the entries of the adjacency lists as follows: in the adjacency list of each vertex $v \in V$, the neighbors of v having labels not exceeding the label of v occupy the first part of the list and the other neighbors appear in the latter part in nondecreasing order of the labels. Thus all the neighbors of each vertex $u \in U$ appear in the adjacency list of u in nondecreasing order of the labels, so that the first parts of the adjacency lists represent G_k. We also employ the same strategy as the triangle listing algorithm, that is, process the vertices of G_k in the nonincreasing order of degrees in G_k. Thus the procedure is refined as follows.

```
procedure COMPLETE($l$, $G$);
  procedure K($k$, $U$);
   {$U$ is the vertex set of $G_k$. $d_k(v)$ is the degree of vertex $v$ in $G_k$.}
    begin
     if $k=2$
      then
```

```
1:  for each edge $(x, y)$ of the subgraph induced by $U$
      do print out $\{x, y\} \cup C$
    else
     begin
2:    sort the vertices in $U$ in way that $d_k(v_1) \geqq d_k(v_2) \geqq \cdots \geqq d_k(v_{|U|})$, and store
      them in list;
      for $i := 1$ to $|U|$
       do begin
          let $U'$ be the set of all the vertices which are adjacent to $v_i$ and labeled
          "$k$"; {$U'$ is the vertex set of $G_{k-1}$.}
3:        relabel all the vertices in $U'$ "$k-1$";
4:        in the adjacency list of each vertex $u \in U'$, move the neighbors of $u$ in
          $U'$ at the first part; {the vertices of $G_{k-1}$ occupy the first parts of the
          adjacency lists of vertices in $U'$, which realize the adjacency lists of
          $G_{k-1}$.}
5:        determine the degree $d_{k-1}(u)$ of each $u \in U'$ in $G_{k-1}$;
6:        add the vertex $v_i$ to $C$;
7:        K$(k-1, U')$;
8:        delete the top entry $v_i$ from $C$;
9:        relabel all the vertices in $U'$ "$k$"; {recovery to $G_k$}
10:       relabel $v_i$ "$k+1$"; {logical (not physical) deletion of $v_i$ from $G_k$}
11:       in the adjacency list of each vertex $v \in U'$, move the entry $v_i$ to the
          position next to the last entry containing a vertex labeled "$k$";
        end
     end
end;
begin {of COMPLETE}
  label all the vertices of $G$ "$l$";
  determine $d_l(v)(=d(v))$ for each $v \in V$;
  $C := \varnothing$;
  K$(l, V)$ {$V$ is the vertex set of $G = G_l$}
end {of COMPLETE};
```

We have the following result on the algorithm.

THEOREM 3. *If a connected graph G has n vertices and m edges, then Algorithm* COMPLETE *lists all the complete subgraphs of order l ($\geqq 2$) in G in $O(la(G)^{l-2}m)$ time using linear space.*

Proof. (a) *Correctness.* Note that throughout the execution of COMPLETE the entries of the adjacency lists are ordered as mentioned just before the refined algorithm. then one can easily verify the correctness of the refined one as well as the original one.

(b) *Space.* We use the same data structure as the algorithm K3 to represent a graph. One recursive call with respect to a vertex v produces a list which stores the vertices in U in nonincreasing order of degree in G_k. The length of the list is at most $d(v)$. Therefore the total length over all the lists with respect to the vertices in C is at most $\sum_{v \in C} d(v) \leqq 2m$ during the execution of the algorithm. Thus the algorithm requires linear space.

(c) *Time.* We now establish the claim on the running time. If the subgraph G_k induced by U has m edges and n vertices, let $T(k, m, n)$ be the time required by procedure K(k, U) to *find* all the K_k in G_k. Here $T(k, m, n)$ does not count the time required by printing out K_l in Statement 1. First consider the case $k=2$, in which Statement 1 is executed. One can find all the edges of G_k in $O(m+n)$ time, because the edges of G_k occupy the first parts of the adjacency lists. Thus Statement 1 requires at most $O(m+n)$ time, and so $T(2,m,n)=O(m+n)$. Next consider the case $k\geqq 3$. Clearly Statement 2 can be executed in $O(n)$ time. Consider the time required by the ith iteration of the **for** statement. Statements 3 and 9 require $O(d_k(v_i)+1)$ time, and Statements 6, 8, and 10 require $O(1)$ time. Just before Statement 3 is executed, in the adjacency list of each $u\in U$, the neighbors of u appear in nondecreasing order of the labels, which are "k", "$k+1$", $\cdots$, "l". Therefore Statement 4 is performed as follows: in the adjacency list of each $u\in U'$, choose the vertices in U' (labeled "$k-1$") among the first $d_k(u)$ entries; and move them to the first part of the list. Thus Statement 4 requires $O(\sum_{u\in U'}(d_k(u)+1))$ time. Similarly one can show that Statements 5 and 11 require $O(\sum_{u\in U'}(d_k(u)+1))$ time. Statement 7 requires $T(k-1, (\sum_{u\in U'} d_{k-1}(u))/2, d_k(v_i))$ time by the definition of T. Note that the graph G_{k-1} induced by U' has at most $(\sum_{u\in U'} d_{k-1}(u))/2$ edges and $d_k(v_i)$ vertices. Thus, the ith iteration of the **for** statement requires

$$O(d_k(v_i))+O\left(\sum_{u\in U'} d_k(u)\right)+O(1)+T\left(k-1,\left(\sum_{u\in U'} d_{k-1}(u)\right)\Big/2, d_k(v_i)\right)$$

time. Each $v_i\in U$ satisfies $d_k(v_i)\geqq d_k(u)$ for every $u\in U'$. Therefore Lemma 2 implies that

$$\sum_{v_i\in U}\left\{O(d_k(v_i))+O\left(\sum_{u\in U'} d_k(u)\right)+O(1)\right\}=O(a(G_k)m+n).$$

Thus we have the recurrence

$$T(2,m,n)=O(m+n),$$

$$T(k,m,n)\leqq O(a(G_k)m+n)+\sum_{v_i\in U} T\left(k-1,\left(\sum_{u\in U'} d_{k-1}(u)\right)\Big/2, d_k(v_i)\right).$$

Solving the reccurence with noting $a(G_{k-1})\leqq a(G_k)$, we have $T(k,m,n)=O(a(G_k)^{k-2}m+n)$.

Since procedure COMPLETE (l, G) calls K(k, U) with $k=l$ and $U=V$ for a connected graph $G=(V,E)$, it requires $O(a(G)^{l-2}m)$ time in total to find all the K_l in G. This fact implies that the number of K_l in G is at most $O(a(G)^{l-2}m)$. Since one can print out a K_l in $O(l)$ time, the total running time of COMPLETE including the time for printing is at most $O(la(G)^{l-2}m)$. Q.E.D.

Theorem 3 together with Lemma 1(c) imply that Algorithm COMPLETE lists all the K_4 in a planar graph in *linear* time. The time complexity is the same as the algorithm of Papadimitriou and Yannakakis [9], but our algorithm does not need the plane embedding of a graph.

6. Clique listing algorithm.

Tsukiyama et al. [11] presented an algorithm MIS which lists all the maximal independent sets in a graph G and requires $O(mn)$ time per maximal independent set. In this section, we first show that our strategy can reduce the running time to $O(a(G)m)$. Then, employing their idea and our strategy, we present an algorithm which lists all the cliques in a graph G in $O(a(G)m)$ time per clique.

The algorithm of Tsukiyama et al. is outlined as follows. Let $G=(V,E)$ be a given graph with vertex set $V=\{1,2,\cdots,n\}$. Each vertex is referred by the number. Let G_i, $1\leqq i\leqq n$, be the subgraph of G induced by vertices $1, 2, \cdots, i$. $N(i)$ denotes the set of vertices adjacent to i in the given graph G. Assume that I_{i-1} is a maximal independent set of G_{i-1}, then one can decide by the following rules whether I_{i-1} or $(I_{i-1}-N(i))\cup\{i\}$ is a maximal independent set in G_i:

(1) If $I_{i-1}\cap N(i)\neq\emptyset$, then I_{i-1} is a maximal independent set of G_i.

(2) If there is no independent set I of G_{i-1} such that $I-N(i)\supsetneq I_{i-1}-N(i)$, then $(I_{i-1}-N(i))\cup\{i\}$ is a maximal independent set of G_i.

Thus they recursively generate all the maximal independent sets of G_i from the maximal independent sets of G_{i-1}. However duplications may occur in maximal independent sets produced by rule (2), so they avoided the duplications by choosing the lexicographically largest one among all the independent sets I_{i-1} having the same $I_{i-1}-N(i)$.

Tsukiyama et al. [11] implemented the backtracking algorithm MIS in a way that one recursive step on vertex i is performed in $O(\sum_{x\in N(i)-\{i+1,\cdots,n\}}d(x))=O(m)$ time, so that MIS requires $O(mn)$ time to find one maximal independent set. An easy observation leads us to an algorithm which requires $O(a(G)m)$ time per maximal independent set. We simply number the vertices of a given graph G in such a way that $d(1)\leqq d(2)\leqq\cdots\leqq d(n)$, and apply the same recursive method. Then, applying Lemma 2, we can easily show that the new algorithm requires

$$O\left(\sum_{1\leqq i\leqq n}\sum_{x\in N(i)-\{i+1,\cdots,n\}}d(x)\right)\leqq O\left(\min_{(u,v)\in E}\{d(u),d(v)\}\right)=O(a(G)m)$$

time per maximal independent set. Unlike the preceding three algorithms, we number the vertices in nondecreasing order of degree so that the newly added vertex i has the largest degree in G_i. If G is sparse, the time complexity $O(a(G)m)$ is considerably better than $O(mn)$.

The problem of listing all the cliques of a graph G is equivalent to that of listing all the maximal independent sets of the complement G^c of G. Therefore the algorithm suggested above can list all the cliques of a graph G in $O(a(G^c)m^c)$ time per clique, where $m^c=n(n-1)/2-m$ is the number of edges of G^c. However, this algorithm is not necessarily efficient for sparse graphs. Using a recursive method similar to MIS, we next give an algorithm CLIQUE which lists all the cliques in $O(a(G)m)$ time per clique. Unlike the case of maximal independent sets, guaranteeing the time complexity of $O(a(G)m)$ is not straightforward in this case, but requires some nontrivial arguments especially on the "lexico. test".

The set of vertices in a clique C is also denoted by C. The following is the outline of our algorithm CLIQUE.

```
procedure CLIQUE
  procedure UPDATE ($i$, $C$)
    {generate a new clique of $G_i$ from a clique $C$ of $G_{i-1}$.}
    begin
      if $i=n+1$
        then print out a new clique $C$   {$C$ is a clique of $G=G_n$.}
        else
          begin
            if $C-N(i)\neq\emptyset$ then UPDATE ($i+1$, $C$); {$C$ is a clique of $G_i$.}
            if both "maximality test" and "lexico. test" succeed
              then
                begin
                  $SAVE:=C-N(i)$;          {save the vertices removed from current $C$}
                  $C:=(C\cap N(i))\cup\{i\}$;  {new $C$ is a clique of $G_i$.}
                UPDATE $(i+1, C)$;
                $C := (C - \{i\}) \cup SAVE$ {recovery to old $C$}
            end
        end
    end;
begin
    number the vertices of a given graph $G$ in such a way that $d(1) \leqq d(2) \leqq \cdots \leqq$
        $d(n)$;
    $C := \{1\}$; {$C$ is the unique clique of $G_1$.}
    UPDATE (2, $C$)
end;
```

In the algorithm above, "maximality test" checks whether the candidate of a new clique $C' = (C \cap N(i)) \cup \{i\}$ is indeed a clique (i.e. maximal complete subgraph) of G_i. The "lexico. test" checks whether C is the lexicographically largest clique of G_{i-1} containing $C \cap N(i)$ $(= C_0)$. This test avoids the duplications of cliques. Note that the same clique C' of G_i may be produced more than once from distinct cliques of G_{i-1} containing C_0. One can easily verify the correctness of the algorithm CLIQUE by induction on n. In what follows, we refine the algorithm so that it runs in $O(a(G)m)$ time per clique.

We begin with the following lemma, which implies that if a clique of G_i is generated from a clique C of G_{i-1} in $O(\sum_{x \in C} d(x))$ time, then one clique of G can be found in $O(a(G)m)$ time.

LEMMA 3. *Let the vertices* $1, 2, \cdots, n$ *of a graph G satisfy* $d(1) \leqq d(2) \leqq \cdots \leqq d(n)$, *and let* C_i, $1 \leqq i \leqq n$, *be an arbitrary clique of* G_i *where* $G = G_n$. *Then*

$$\sum_{1 \leqq i \leqq n} \sum_{x \in C_i} d(x) \leqq 4a(G)m.$$

Proof. Let $c = \max_{1 \leqq i \leqq n} |C_i|$, then Equation (2) implies that

$$c \leqq 2a(K_c) \leqq 2a(G). \tag{3}$$

Since $d(i) \geqq d(x)$ for any $x \in C_i$,

$$\sum_{1 \leqq i \leqq n} \sum_{x \in C_i} d(x) \leqq \sum_{1 \leqq i \leqq n} \sum_{x \in C_i} d(i) \leqq \sum_{1 \leqq i \leqq n} d(i)c \leqq 2mc.$$

Combining this with (3), we have

$$\sum_{1 \leqq i \leqq n} \sum_{x \in C_i} d(x) \leqq 4a(G)m.$$ Q.E.D.

The following three lemmas are concerned with the tests.

LEMMA 4 [maximality test]. *Let C be a clique of* G_{i-1}. *Then,* $(C \cap N(i)) \cup \{i\}$ *is a clique of* G_i *if and only if* G_i *has no vertex* $y \in N(i) - C$ *such that* $y < i$ *and* $N(y) \supset C \cap N(i)$.

Proof. Immediate. Q.E.D.

Using Lemma 4, one can perform the "maximality test" once in $O(d(i) + \sum_{x \in C \cap N(i)} d(x))$ time as follows: first compute $T(y) = |N(y) \cap C \cap N(i)|$ for $y \in V$ (in that time); then check whether there exists $y \in N(i) - C$ such that $y < i$ and $T(y) = |C \cap N(i)|$. (We will describe the detail later in the refined algorithm CLIQUE.)

LEMMA 5. *Let* C_0 *be a complete subgraph of a graph G. A clique* $C(\supset C_0)$ *of G is the lexicographically largest one among all the cliques containing* C_0 *if and only if there is no vertex* $y \notin C$ *such that* $N(y) \supset C_0 \cup C^y$, *where* $C^y = \{k \in C \mid k > y\}$.

Proof. Necessity. Assume that there exists a vertex $y \notin C$ such that $N(y) \supset C_0 \cup C^y$. Then clearly there exists a clique containing $\{y\} \cup C_0 \cup C^y$ which is lexicographically larger than C.

Sufficiency. Assume that there exists a clique $C' \supset C_0$ which is lexicographically larger than C. Let y be the largest vertex in $C' - C$. Then $C \cap C' \supset C^y$ since every vertex in $(C - C')$ is less than y. Thus we have $N(y) \supset C \cap C' \supset C_0 \cup C^y$. Q.E.D.

The direct application of Lemma 5 would require $O(m)$ time to perform the "lexico. test" once, so the algorithm would require $O(mn)$ time per clique. The following lemma yields a more efficient "lexico. test".

LEMMA 6 [lexico. test]. *Let C be a clique of G which includes a complete subgraph C_0, where C_0 may be empty. Let $p = |C - C_0|$, let $j_1 < j_2 < \cdots < j_p$ be the vertices in $C - C_0$, and let $j_0 = 0$. For each vertex $y \notin C$, let $S(y) = |N(y) \cap (C^y - C_0)|$, and let $j_k > y$ be the smallest vertex in $N(y) \cap (C^y - C_0)$ if $S(y) \geqq 1$. Then C is the lexicographically largest clique containing C_0 if and only if every $y \notin C$ such that $N(y) \supset C_0$ satisfies*

(a) *if $S(y) \geqq 1$ then either $S(y) + k - 1 < p$ or $j_{k-1} > y$;*

(b) *if $S(y) = 0$ then $j_p > y$.*

Proof. Necessity. Assume that there exists a vertex $y \notin C$ such that $N(y) \supset C_0$, violating either (a) or (b). If $S(y) = 0$ and $j_p < y$, then $C^y = \emptyset$ and there exists a clique which includes $\{y\} \cup C_0$ and is lexicographically larger than C. Thus we may assume that $S(y) \geqq 1$, $S(y) + k - 1 = p$ and $j_{k-1} < y$. (Note that $S(y) + k - 1 \leqq p$.) Then the inequality $j_{k-1} < y$ implies $C^y - C_0 = \{j_k, j_{k+1}, \cdots, j_p\}$, so $|C^y - C_0| = p - k + 1$. Combining this with $S(y) + k - 1 = p$, we have $S(y) = |C^y - C_0|$. Therefore there exists a clique which includes $\{y\} \cup C^y \cup C_0$ and is lexicographically larger than C.

Sufficiency. Assume that there exists a clique $C' (\supset C_0)$ which is lexicographically larger than C. Let y be the largest vertex in $C' - C$. Then we have $N(y) \supset C^y \cup C_0$ as shown in the proof of Lemma 5. If $S(y) = 0$, then clearly $j_p < y$, violating (b). Thus we may assume that $S(y) \geqq 1$. Then clearly $j_{k-1} < y$ and $S(y) = |C^y - C_0|$, so $S(y) + k - 1 = |C^y - C_0| + k - 1 = p$, violating (a). Q.E.D.

Using Lemma 6, one can perform the "lexico. test" once in $O(\sum_{x \in C} d(x))$ time. We first compute $|N(y) \cap (C - C_0)|$ for $y \in V - C$ and then alter them to $S(y) = |N(y) \cap (C^y - C_0)|$, as shown in the refined CLIQUE. Thus the computation of $S(y)$ requires $O(\sum_{x \in C - C_0} d(x))$ time. Let $G = G_{i-1}$ as in the algorithm, then the direct access of the vertices $y \notin C$ such that $N(y) \supset C_0$ $(= C \cap N(i))$ would require $O(i)$ time, which may be greater than $O(\sum_{x \in C} d(x))$. However, we can perform the access in $O(\sum_{x \in C} d(x))$ time as follows. If either $C_0 \neq \emptyset$ or $S(y) \geqq 1$, then y is accessible from the adjacency lists of vertices in C. On the other hand, if $C_0 = \emptyset$ and $S(y) = 0$, then y is not accessible from these lists. However, if (i) $C_0 = \emptyset$, (ii) C is not the lexicographically largest clique containing C_0 in G_{i-1}, and (iii) every $y \notin C$ satisfies condition (a) of Lemma 6, then C does not contain the largest vertex $i - 1$ of G_{i-1}. (Consider the largest clique C' and the largest vertex y in $C' - C$.) Thus in this case we can perform the "lexico. test" simply by checking whether C contains vertex $i - 1$, as will be known in the algorithm.

We are now ready to present the refined algorithm CLIQUE.

```
procedure CLIQUE;
  procedure UPDATE ($i$, $C$);
   begin
    if $i = n + 1$
     then print out a new clique $C$
     else
      begin
```

```
1:    if $C - N(i) \neq \emptyset$ then UPDATE $(i+1, C)$;
    {prepare for tests}
      {compute $T[y] = |N(y) \cap C \cap N(i)|$ for $y \in V - C - \{i\}$}
2:    for each vertex $x \in C \cap N(i)$
        do for each vertex $y \in N(x) - C - \{i\}$
            do $T[y] := T[y] + 1$;
      {compute $S[y] = |N(y) \cap (C - N(i))|$ for $y \in V - C$}
3:    for each vertex $x \in C - N(i)$
        do for each vertex $y \in N(x) - C$
            do $S[y] := S[y] + 1$;
      $FLAG := true$;
    {maximality test}
4:    if there exists a vertex $y \in N(i) - C$ such that $y < i$ and $T[y] = |C \cap N(i)|$
        then $FLAG := false$; {$(C \cap N(i)) \cup \{i\}$ is not a clique of $G_i$}
    {lexico. test}
      {$C \cap N(i)$ corresponds to $C_o$ in Lemma 6}
5:    sort all the vertices in $C - N(i)$ in ascending order $j_1 < j_2 < \cdots < j_p$, where
          $p = |C - N(i)|$;
      {case $S(y) \geqq 1$. See Lemma 6.}
6:    for $k := 1$ to $p$
        do for each vertex $y \in N(j_k) - C$ such that $y < i$ and $T[y] = |C \cap N(i)|$
            do if $y \geqq j_k$
                then $S[y] := S[y] - 1$ {alter $S[y]$ to $S(y)$}
                else
                  if ($j_k$ is the first vertex which satisfies $y < j_k$)
                    then {$S[y] = S(y)$}
                      if $(S[y] + k - 1 = p)$ and $(y \geqq j_{k-1})$ {$j_0 = 0$}
                        then $FLAG := false$; {$C$ is not lexico. largest}
      {case $S(y) = 0$}
7:    if $C \cap N(i) \neq \emptyset$
        then for each vertex $y \notin C \cup \{i\}$ such that $y < i$, $T[y] = |C \cap N(i)|$ and
                $S[y] = 0$
                  {access $y$ from the adjacency list of a vertex in $C \cap N(i)$}
            do if $j_p < y$ then $FLAG := false$        {$C$ is not lexico. largest.}
        else if $j_p < i - 1$ then $FLAG := false$;     {$C$ is not lexico. largest.}
    {reinitialize $S$ and $T$}
8:    for each vertex $x \in C \cap N(i)$
        do for each vertex $y \in N(x) - C - \{i\}$
            do $T[y] := 0$;
9:    for each vertex $x \in C - N(i)$
        do for each vertex $y \in N(x) - C$
            do $S[y] := 0$;
    {$FLAG$ is true if and only if $(C \cap N(i)) \cup \{i\}$ is a clique of $G_i$ and $C$ is the
    lexicographically largest clique of $G_{i-1}$ containing $C \cap N(i)$.}
10:   if $FLAG$
        then
          begin
            $SAVE := C - N(i)$;
            $C := (C \cap N(i)) \cup \{i\}$;
            UPDATE $(i+1, C)$;
```

```
                $C := (C - \{i\}) \cup SAVE$
            end
        end
    end;
  begin {of CLIQUE}
    number the vertices of a given graph $G$ in such a way that $d(1) \leqq d(2) \leqq \cdots \leqq$
        $d(n)$;
    for $i := 1$ to $n$ {initialize $S$ and $T$}
      do begin $S[i] := 0$; $T[i] := 0$ end;
    $C := \{1\}$;
    UPDATE $(2, C)$
  end {of CLIQUE};
```

We have the following theorem.

THEOREM 4. *Algorithm* CLIQUE *lists all the cliques of a connected graph G in $O(a(G)m)$ time per clique, using $O(m)$ space.*

Proof. Using Lemmas 4 and 6, one can prove the correctness. Therefore we shall concentrate on the claim on time and space.

Let C_n be an arbitrary clique of $G = G_n$, and inductively define C_i, $n-1 \geqq i \geqq 1$, to be the clique of G_i from which C_{i+1} is generated by procedure CLIQUE.

Consider the time $T(i)$ required by UPDATE (i, C_{i-1}), excluding the time required by the recursive calls in Statements 1 and 10. Noting the remark mentioned just before the refined CLIQUE, one can easily show that all the Statements 1-10 except 5 can be executed in $O(d(i) + |C_{i-1}| + \sum_{x \in C_{i-1}} d(x))$ time. We now show that the sorting in Statement 5 also requires at most $O(\sum_{x \in C_{i-1}} d(x))$ time. One can sort p items in $O(p \log p)$ time where $p = |C_{i-1} - N(i)|$ [1]. Since the subgraph induced by $C_{i-1} - N(i)$ is a complete subgraph, $O(p \log p) \leqq O(p(p-1)) \leqq O(\sum_{x \in C_{i-1} - N(i)} d(x))$. Here the bucket sort should not be used, because it requires $O(j_p)$ time, which may be greater than $O(p \log p)$. Thus $T(i) \leqq O(d(i) + |C_{i-1}| + \sum_{x \in C_{i-1}} d(x))$.

Hence the total time required to generate C_n is at most $\sum_{2 \leqq i \leqq n} T(i) \leqq O(\sum_{2 \leqq i \leqq n} (d(i) + |C_{i-1}| + \sum_{x \in C_{i-1}} d(x)))$. Lemma 3 implies that the time is $O(a(G)m)$.

Every UPDATE (i, C), $i \leqq n$, calls at least once UPDATE $(i+1, C)$ in Statement 1 or 10. In fact, if the recursive call does not occur in Statement 1, then it necessarily occurs in Statement 10. Thus every call of UPDATE eventually generates at least one clique, and hence the time spent by any statement is counted in the time above at least once for some clique C_n of G_n. Thus we have shown that CLIQUE requires $O(a(G)m)$ time per clique.

Since set C is a global variable, C requires $O(n)$ space. Since the sets of vertices contained in the local variable *SAVE* are pairwise disjoint, *SAVE* requires $O(n)$ space in total. The arrays S, T and the adjacency lists require $O(m)$ space. Thus CLIQUE requires $O(m)$ space in total. Q.E.D.

7. Conclusion.

In this paper we introduced a simple edge-searching strategy and presented the four efficient algorithms for the various subgraph listing problems. We used the arboricity $a(G)$, a rather unfamilar graph invariant, as a parameter in bounding the running time of algorithms. Our algorithms are as fast as the previous ones if any, and a factor n is often reduced to $a(G)$ in the running time. The key idea is in Lemma 2, which implies that if a certain operation on a graph consumes $O(\min\{d(u), d(v)\})$ time for each edge (u, v) then the operation can be executed for all the edges in a graph G in $O(a(G)m)$ time. It is expected that this result will find a number of other applications in graph problems.

Finally we remark that in this paper only the concept of arboricity is used in the analysis of the running time of algorithms and that any of our algorithms requires neither to find $a(G)$ nor to decompose a graph into the minimum number of edge-disjoint forests.

REFERENCES

[1] A. V. AHO, J. E. HOPCROFT AND J. D. ULLMAN, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.

[2] J. G. AUGUSTON AND J. MINKER, *An analysis of some graph theoretical cluster techniques*, J. Assoc. Comput. Mach., (1970), pp. 571-588.

[3] R. BAR-YEHUDA AND S. EVEN, *On approximating a vertex cover for planar graphs*, Proc. 14th Annual ACM Symposium on Theory of Computing, San Francisco, May 5-7, 1982, pp. 303-309.

[4] N. CHIBA, T. NISHIZEKI AND N. SAITO, *An algorithm for finding a large independent set in planar graphs*, Networks, 13 (1983), pp. 247-252.

[5] F. HARARY, *Graph Theory*, revised, Addison-Wesley, Reading, MA, 1972.

[6] A. ITAI AND M. RODEH, *Finding a minimum circuit in a graph*, this Journal, 7, (1978), pp. 413-423.

[7] C. ST. J. A. NASH-WILLIAMS, *Edge-disjoint spanning trees of finite graphs*, J. London Math. Soc., 36 (1961), pp. 445-450.

[8] T. NISHIZEKI, T. ASANO AND T. WATANABE, *An approximation algorithm for the Hamiltonian walk problem on a maximal planar graph*, Discr. Appl. Math., 5 (1983), pp. 211-222.

[9] C. H. PAPADIMITRIOU AND M. YANNAKAKIS, *The clique problem for planar graphs*, Inform. Proc. Lett. 13, 4, 5 (1981), pp. 131-133.

[10] R. C. READ AND R. E. TARJAN, *Bounds on backtrack algorithmms for listing cycles, paths, and spanning trees*, Networks, 5 (1975), pp. 237-252.

[11] S. TSUKIYAMA, M. IDE, H. ARIYOSHI AND I. SHIRAKAWA, *A new algorithm for generating all the maximal independent sets*, this Journal, 6, (1977), pp. 505-517.