Linear Algorithms to Recognize Interval Graphs and Test for the Consecutive Ones Property

by

Kellogg S. Booth†
Department of Electrical Engineering
and Computer Sciences
University of California, Berkeley
and
Lawrence Livermore Laboratory

George S. Lueker‡
Department of Electrical Engineering
and Program in Applied Mathematics
Princeton University

Abstract

A matrix of zeroes and ones is said to have the consecutive ones property if there is a permutation of its rows such that the ones in each column appear consecutively. This paper develops a data structure which may be used to test a matrix for the consecutive ones property, and produce the desired permutation of the rows, in linear time. One application of the consecutive ones property is in recognizing interval graphs. A graph is an interval graph if there exists a 1-1 correspondence between its vertices and a set of intervals on the real line such that two vertices are adjacent if and only if the corresponding intervals have a nonempty intersection. Fulkerson and Gross have characterized interval graphs as those for which the clique versus vertex incidence matrix has the consecutive ones property. In testing this particular matrix for the consecutive ones property we may process the columns in a special order to simplify the algorithm. This yields the interval graph recognition algorithm which is presented in section 2; section 3 indicates how this algorithm may be extended to the general consecutive ones problem.

A final section of the paper gives a number of further applications of the ideas developed in the earlier sections. These applications include linear algorithms to

a) recognize unit interval graphs,

b) test for the circular ones property,

c) recognize planar graphs,

d) count the number of distinct models of an interval graph (assuming that an arithmetic operation can be done in constant time), and

e) determine whether two interval graphs are isomorphic.

1. Introduction.

Let A be a matrix all of whose elements are zeroes and ones. Let n be the larger dimension of A, and let f be the number of ones in A. Assume that the matrix is represented by a set of lists, one for each column, specifying which rows contain ones. Note that the size of such a representation is $O(n+f)$; we will say an algorithm which takes A as input is linear if it runs in $O(n+f)$ time. The matrix A is said to have the consecutive ones property if its rows may be permuted in such a way as to

† Research performed under the auspices of the Atomic Energy Commission/Energy Resources Development Agency.

‡ Research supported by a National Science Foundation Graduate Fellowship and by NSF Grant GJ-1052.

make the ones in each column consecutive [FG]. Such matrices are related to problems in a number of areas, including archaeology [Ke] and information retrieval [G]. Tucker [T72] has presented a structure theorem for these matrices.

One of the most interesting applications of the consecutive ones property, however, is in characterizing interval graphs. A graph G is an ordered pair (V,E), where V is a set whose elements are called vertices, and E is a set of unordered pairs of distinct elements of V, called edges. Vertices v and w are adjacent if {v,w} is an element of E. The statement that v is adjacent to w will sometimes be abbreviated as

v adj w.

Let $n = |V|$ and $e = |E|$. G is assumed to be represented by a set of n lists, one for each vertex, with the list for a vertex v containing all vertices adjacent to v. The size of such a representation is $O(n+e)$; a graph algorithm will be called linear if it requires $O(n+e)$ time. A graph is said to be an interval graph if there exists a 1-1 correspondence between its vertices and a set of intervals on the real line such that two vertices are adjacent if and only if the corresponding intervals have a nonempty intersection. The set of intervals will be called a model for G. It appears that Hajos [H] was the first to introduce interval graphs into the literature. Since then interval graphs have been related to problems in various fields, including biology [B], psychology [R68, R69], traffic light sequencing [St], and ecology [C]; a summary of various applications is given in [R].

A graph is said to be chordal if every cycle of length greater than or equal to four has a chord; it is known that all interval graphs are chordal [LB]. If we know that a graph G is chordal, various authors have provided additional conditions which make it an interval graph. [LB] show that G is an interval graph if and only if G is chordal and not asteroidal; they present a test to determine whether G is asteroidal, and give an $O(n^3)$ time bound for this test. From the results of [GH], we know that G is an interval graph if and only if in addition to being chordal, it has a transitively orientable complement $\bar{G}$; they present an algorithm to test whether $\bar{G}$ is transitively orientable, but give no time bound. (We know of no linear implementation of their test.) Another algorithm to test whether a graph is transitively orientable is given in [PLE]; again, no bound is given. The characterization of greatest value for this paper is due to Fulkerson and Gross [FG]. A set of vertices is complete if all of its elements are adjacent to each other; a clique is a maximal complete set of vertices. Form a 0-1 matrix A with a row for each clique and a column for each vertex; put a one (respectively zero) in each position for which the corresponding vertex is (respectively, is not) in the corresponding clique. This is called the clique versus vertex incidence matrix of the graph. [FG] show that G is an interval graph if and only if A has the consecutive ones property. Equivalently, G is an interval graph iff its cliques may be written down in an order such that for each v, elements of C(v) appear consecutively, where C(v) is the set of all cliques that contain v. They give a test for this property which involves the formation of $O(n^2)$ inner products. Although this characterization is sufficient without the additional restriction that G be chordal, [FG] found it useful to exploit the chordality of G to find the cliques in polynomial time. Since then [RTL], [BT], and [L] have shown how to use a technique called lexicographic breadth first search (LBFS) to recognize chordal graphs and determine their cliques in linear time. (LBFS is closely related to an ordering scheme used in [CG] and [Se].) Thus all that remains necessary for the construction of a linear interval graph recognition algorithm is a linear algorithm to test for the consecutive ones property of the clique versus vertex incidence matrix. In the following section we present such an algorithm. This algorithm processes the columns of the matrix in a special order which simplifies matters. In section 3 we indicate how the algorithm may be extended to handle the general consecutive ones problem.

2. Interval graph recognition.

The method used here to recognize interval graphs is closely related to the method used in [LEC] to recognize planar graphs; more will be said about this later. Our algorithm centers around a structure called a PQ-tree. This is an ordered tree whose nodes fall into three classes, namely the P-nodes, Q-nodes, and C-nodes, such that the leaves are precisely the C-nodes. The C-nodes will be in one-to-one correspondence with the cliques of G, and will henceforth be identified with them. A lower case letter p (respectively q), possibly with superscripts or subscripts, will be used to refer to a P-node (respectively Q-node); when the class of a node is not known, a lower case t is used. The PQ-trees used

during the recognition algorithm will satisfy certain additional properties, namely, that

(2.1) The root is a Q-node.

(2.2) All children of Q-nodes are P-nodes, and all children of P-nodes are Q-nodes or C-nodes.

In order to illustrate PQ-trees, certain conventions will be useful. C-nodes will be denoted by small dots, P-nodes will be denoted by circles, and Q-nodes will be denoted by squares. Trees will always be drawn with the root at the top. An example of a PQ-tree is shown in Figure 2.1. Two nodes are said to be siblings if they have the same parent. Two P-nodes are immediate siblings if they have the same parent and are next to each other. A P-node is endmost if it is leftmost or rightmost among its siblings; otherwise it is interior. If p and p' are siblings, p-p' denotes the sequence of siblings beginning with p and ending with p'.

The frontier of a PQ-tree T, written F(T), is obtained by reading its leaves from left to right; this coincides with the usual definition of frontier. The frontier of a node t in a tree, written F(t), is the frontier of the subtree rooted at t. Two trees T and T' are said to be equivalent if one may be obtained from the other by applying any combination (possibly none) of the following two classes of transformations, called equivalence transformations:

a) arbitrarily reordering the children of a P-node.

b) reversing the children of a Q-node.

Since we will often refer to strings of cliques, it will be convenient to use capital letters near the end of the alphabet to represent such strings. A clique order Z is said to be consistent with T if there is a tree T' such that T' is equivalent to T and Z = F(T'). The set of all orders consistent with T is called the consistent set of T and denoted cs(T).

The universal PQ-tree for a set of k cliques $C_1, C_2, \ldots, C_k$ is the tree with a root q which has one child p which in turn has all k cliques as children. It follows directly from the definition that all clique orders are consistent with this tree. See Figure 2.2.

The recognition algorithm to be described here works by initially setting up the universal PQ-tree for all the cliques and then enforcing, for each v, the condition that those cliques which contain v be consecutive in all elements of cs(T). This condition is imposed by executing a routine REDUCE(C(v)) which modifies T. Assume the call REDUCE(S) is made, where S is the set of cliques to be made consecutive. Node t is said to be empty if its frontier contains no elements of S and pertinent if its frontier contains at least one element of S. Pertinent nodes are further classified as full or partial, according as their frontier is composed entirely or only partly of elements of S.

Several primitive operations will be useful in describing the transformation REDUCE. The first primitive operation to be defined is the function SCAN(p), which returns a P-node, say p'. First it is determined whether p has a pertinent immediate sibling. If it has none, p' is set equal to p. If p has a pertinent immediate sibling on its left (respectively right), the routine sets p' equal to the leftmost (respectively rightmost) pertinent sibling of p such that all siblings between p and p' are full. The routine derives its name from the fact that this can be done by a simple scan. Note that if p has two pertinent immediate siblings, p' is not uniquely determined. This routine is illustrated in Figure 2.3.

A second primitive operation is SPLIT(p,p*). Node p is input to this routine; p* is a new node created by the routine, which is made an immediate sibling of p. All full children of p are moved to p*. It must be determined on which side of p to place p*; this is done as follows. If p has a pertinent immediate sibling on one side, p* is inserted on that side; otherwise, if p is leftmost (respectively rightmost), then p* is inserted on the left (respectively right) of p; whenever SPLIT is called, at least one of these conditions will hold. This routine is illustrated in Figure 2.4.

TRIM(p) deletes a P-node p from the tree; it is used to eliminate nodes with no children, to enforce the condition that the leaves are C-nodes.

COLLAPSE(p,q,p*) is another primitive transformation. Whenever this routine is called, q will have one full endmost child and one empty endmost child. Node q is deleted from the tree and its children are made children of the parent of p; they are inserted between p and p* in the same order they appeared as children of q, except that they are reversed if necessary so that the new immediate sibling of p*

is full. This is illustrated in Figure 2.5.

The final primitive operation to be introduced is EXTEND(p,q,p*). Here p and q are input and p* is the name of a new node which may be created by the routine. If p has no full children, the routine simply returns. Otherwise all full children of p are taken from p and made children of a newly created node p*. When this routine is called, q will either be null or have a unique full endmost, say leftmost (respectively rightmost), child; in the latter case, p* is made an immediate sibling of this child, on its left (respectively right). If q has the value null instead of the name of a Q-node, a new Q-node is created, named q, and made the child of p; p* becomes the only child of q. Figure 2.6 illustrates this.

Before actually stating the algorithm, we need to introduce the idea of a descending degree lexicographic breadth first search order (DDLBFS-order) of V. (This is a restricted form of the order used in [RTL, BT, and L].) Two vertices v and w are said to agree on a vertex x if

$$v \text{ adj } x \iff w \text{ adj } x;$$

otherwise, they disagree on x. If we write the vertices of V down in some left to right order, we say it is a DDLBFS-order if for v and w in V, with v to the left of w, either

a) v and w agree on all vertices to the left of v and v has degree greater than or equal to the degree of w, or

b) the leftmost vertex to the left of v on which v and w disagree is adjacent to v.

The statement v is to the left of w will sometimes be written $v < w$. It will be explained later why it is useful to process the vertices in a DDLBFS-order.

The complete algorithm is given on the following page. Figure 2.7 illustrates the REDUCE routine.

Lemma 2.1. The modifications to T by REDUCE never add any new consistent orders.

Proof. One easily sees that SPLIT, COLLAPSE, and EXTEND add no new orders. []

Lemma 2.2. If a call to REDUCE(S) does not reject, then after the call any order in cs(T) has all elements of S consecutive.

Proof sketch. One can show that after the call some node q in T has a consecutive sequence of children whose children in turn comprise precisely S. []

Lemma 2.3. Each time the call REDUCE(S) is executed, the following two conditions are satisfied:

(2.3) S does not properly contain the frontier of any Q-node.

(2.4) There exists a node t such that all partial Q-nodes are ancestors of t.

Proof sketch. If (2.3) were violated, we could show that there were two vertices v and w such that $v < w$ in the DDLBFS-order, but $C(v) \subset C(w)$. This is easily shown to contradict the definition of a DDLBFS-order.

If (2.4) were violated, we could produce a contradiction of the chordality of G, using a theorem in [RTL, BT, and L]; the details are omitted. []

Lemma 2.4. Suppose that before some call to REDUCE(S) in step 34, cs(T) contains some clique order Z which has elements of S consecutive. Then the call does not reject, and after the call Z is still in cs(T).

Proof sketch. If there is a Q-node whose frontier contains precisely S, one sees by inspection that REDUCE does not change cs(T); thus Z is not eliminated. Now assume there is no Q-node whose frontier contains precisely S. Then condition (2.3) shows that the labelling of "full" P-nodes is correct. Condition (2.4), together with the fact that S may be made consecutive in the frontier of a tree equivalent to T, may be used to show that COUNT eventually equals |S| and the routine terminates successfully.

To see that Z is not eliminated from cs(T), we can examine each SPLIT, COLLAPSE, and EXTEND and conclude that they eliminate only orders in which not all elements of S are consecutive. []

Theorem 2.1. Algorithm 1 correctly recognizes interval graphs, and can be implemented to run in $O(n+e)$ time.

Proof sketch. Correctness follows from Lemmas 2.1 through 2.4. To implement the test to run in linear time, note that the test for chordality and determination of the sets C(v) can be done in linear time using the methods of [RTL, BT, and L];

Algorithm 1. Interval Graph Recognition.

```
  begin
     REDUCE: procedure(S);
        begin
 1.        for each P-node p with all of its children in S, label p "full";
 2.        if any endmost P-node has a child in S
 3.           then
                 begin
 4.                 q ← the most recently created Q-node which is a parent
                     of an endmost P-node with a child in S;
 5.                 if q has a full endmost child
 6.                    then let p' be a full endmost child of q
 7.                    else let p' be an endmost child of q such that p' has
                      a child in S;
                 end
 8.           else p' ← any P-node with a child in S;
 9.        p ← SCAN(p'); COUNT ← 0; q ← null;
10.        do forever
              LOOP: begin
11.              set p' ← SCAN(p) and add to COUNT the number of children
                  in S of the nodes in the sequence p-p';
12.              if COUNT = |S| then go to OUT;
13.              if p' is not endmost then reject;
14.              if p' is not full and not equal to p then reject;
15.              SPLIT(p,p*);
16.              if q ≠ null then COLLAPSE(p,q,p*);
17.              q ← parent(p');
18.              if q is the root then reject;
19.              if p has no children then TRIM(p);
20.              if p* has no children then TRIM(p*);
21.              p ← parent(q);
              end;
22.        OUT: if p = p'
23.           then EXTEND(p,q,p*)
              else
                 begin
24.                 SPLIT(p,p*);
25.                 if q ≠ null then COLLAPSE(p,q,p*);
26.                 SPLIT(p',p'*);
27.                 if p has no children then TRIM(p);
28.                 if p* has no children then TRIM(p*);
29.                 if p' has no children then TRIM(p');
30.                 if p'* has no children then TRIM(p'*);
                 end;
        end procedure REDUCE;
31.  place the vertices of G into a DDLBFS-order v1,v2,...,vn;
32.  if G is not chordal then reject;
33.  form the universal PQ-tree for the cliques of G;
34.  for i ← 1 to n do REDUCE(C(vi));
35.  if rejection ever occurred in steps 32 or 34
36.     then write("G is not an interval graph")
37.     else write("G is an interval graph");
  end.
```

the DDLBFS-order can be obtained by a slight modification of the techniques used there.

The linear implementation of REDUCE is mainly a matter of deciding on a good data structure for T. In step 4 we need to decide which of several Q-nodes was most recently created. This can be done by keeping a count of the number of Q-nodes created. When a new Q-node q is created we attach to q an integer telling the total number of Q-nodes created so far; this number is called the creation number of q. To find which of several Q-nodes was most recently created, we merely pick the one with the highest creation number.

The main difficulties, however, arise in COLLAPSE. This operation may cause the children of the deleted Q-node to be reversed; moreover, it changes the parent of each. To make these changes easy to handle, we keep the children of Q-nodes in a modified form of doubly-linked list in which each node has a link to the sibling on its left and right, but the issue of which is which is left open; also, we only require the parent field of a P-node to be valid when the node is endmost. This data structure contains all the information that Algorithm 1 needs, and permits easy modification to reflect the changes made during a call to REDUCE. In fact, the time for a single call is $O(1+k+|C(v_i)|)$, where k is the number of Q-nodes destroyed in the call. Then since the total number of Q-nodes ever created is $O(n)$, the total time in all calls is $O(n+e)$. □

3. The general consecutive ones problem.

When constructing a PQ-tree to test a clique versus vertex matrix for the consecutive ones property, we were able to process the vertices in a special order which guaranteed certain conditions about how the elements of C(v) could be distributed over the frontier of T, as indicated in Lemma 2.3. To solve the general consecutive ones problem, we wish to eliminate the need for such conditions, by developing a more general REDUCE algorithm. In this section we sketch the methods whereby this can be done.

Suppose we are trying to perform the operation REDUCE(S), that is, to modify T in such a way as to eliminate from cs(T) orders in which elements of S do not appear consecutively. Condition (2.3) can be violated since some Q-node may have a frontier which is properly contained in S. Therefore the simple technique used in Algorithm 1 to find full nodes is no longer adequate. However, we can determine which nodes are full by first labelling all leaves in S "full" and then propagating the full labels up the tree whenever some node has all of its children full. This could be quite time-consuming if the labels had to be propagated up a long path of nodes, each of which had only a single child. It is easy to eliminate this possibility, however. We simply enforce the condition that each Q-node must have at least two grandchildren. The tree is then close enough to being binary that the work involved in finding all full nodes can be $O(|S|)$.

A more difficult problem arises out of the fact that condition (2.4) may not be satisfied. Let us say a node is critical if it is partial but has no partial children. It is not hard to see that if S can be made consecutive in the frontier of T, then T has zero, one, or two critical nodes. The rest of this discussion treats the case in which there are two critical nodes, say t_1 and t_2; this is the hardest case. Unfortunately it is not easy to determine which nodes are t_1 and t_2. Creation numbers were used to solve the corresponding problem in Algorithm 1, but this trick is no longer useful here, at least partly because a Q-node is not necessarily generated after its ancestors. One step towards a solution is the observation that t_1 and t_2 have some but not all of their children full. Other nodes will share this property, so the following algorithm may be used to eliminate these other nodes as candidates.

1. Place all nodes with some but not all children full on a queue. Also place them on a list L.

2. Repeat step 3 until the queue contains only one element.

3. Remove a node t from the queue. If its parent t' is in L, remove t' from L. If t' has never been on the queue, add t' to the queue.

If S can be made consecutive, the above algorithm terminates with two elements left in L. These are t_1 and t_2.

Once t_1 and t_2 are known, we modify T by performing two sets of iterations similar to the LOOP of Algorithm 1, one beginning at t_1 and one beginning at t_2. We must know, however, when to stop these iterations and possibly perform an operation similar to the EXTEND of Algorithm 1.

This is determined by propagating counts of descendants in S up the partial nodes until some single node, or consecutive sequence of P-node siblings, is found which accounts for all of S. If all the ideas sketched here are implemented carefully, we obtain the following.

Theorem 3.1. The general consecutive ones problem may be answered in $O(n + f)$ time.

4. Further applications of PQ-trees.

PQ-trees have applications to a number of other problems, which will be briefly described here. First, we will discuss three fast recognition algorithms.

A graph G is a unit inverval graph if there is a model for G in which all intervals have unit length. The class of unit interval graphs is the same as the class of indifference graphs; they are discussed in [R68 and R69]. Let A be the matrix formed by taking the adjacency matrix of G and making all entries along the main diagonal one. It is shown in [R68] that G is a unit interval graph if and only if A has the consecutive ones property. Thus we immediately obtain

Theorem 4.1. Unit interval graphs may be recognized in linear time.

A matrix is said to have the circular ones property if the rows can be permuted so that the ones in each column appear consecutively if we allow them to wrap around from the bottom to the top. Tucker [T71] discusses these in connection with circular-arc graphs; he proposes the following method for testing a matrix A for the circular ones property. Complement (i.e., interchange zeroes and ones) all columns in A which have a one in the first row; call the result A'. Then A has the circular ones property if and only if A' has the consecutive ones property. Let f' be the number of ones in A'. Using the $O(n + f')$ test for the consecutive ones property, we obtain an $O(n^2)$ test for the circular ones property. We can do better, however.

Theorem 4.2. The circular ones property can be decided in $O(n + f)$ time.

Proof. In the above complementing process, instead of using the first row, we use a row which has a minimum number of ones. Then one easily sees that $f' \leq 2f$, so that we have an $O(n + f)$ test for the circular ones property. □

A third application is a linear test for planarity of a graph.

Theorem 4.3. The planarity test of [LEC] may be implemented to run in linear time.

Proof sketch. The heart of the algorithm is the REDUCE operation. The fast techniques for performing this operation, as sketched in section 3, can be used in a linear implementation. □

It should be noted that another linear algorithm to recognize planar graphs has been published by Hopcroft and Tarjan [HT]. Tarjan [Ta] has also given an $O(n^2)$ implementation of the [LEC] algorithm; Even [E] has claimed that the [LEC] algorithm could be implemented to run in linear time, but this has not been published to our knowledge.

PQ-trees also have application to a number of problems other than recognition algorithms. The first of these is an efficient answer to a question discussed in [Ke]. Once a matrix is known to have the consecutive ones property, we may wish to count the number of permutations of its rows which cause the ones in each column to be consecutive. The following theorem gives the answer.

Theorem 4.4. Suppose A has the consecutive ones property and the PQ-tree T is constructed by the recognition algorithm. Then the number of permutations of the rows which make the ones of A consecutive in all columns may be found by taking the product of the following factors over all the P- and Q-nodes t of T:

a) If t is a P-node, let the corresponding factor be the factorial of the number of its children.

b) If t is a Q-node, let the corresponding factor be one if it has only one child, and two otherwise.

The proof is easy and is omitted.

If A is the clique versus vertex incidence matrix of G, then the product calculated in the above theorem is also the number of models of G which differ on strict following, in the sense of [R].

A final application of PQ-trees to be given here is in determining isomorphism of interval graphs. First suppose we drop conditions (2.1) and (2.2) and instead require that all P-nodes have at least two children and all Q-nodes have at least

three children. Call such a tree proper. Given a PQ-tree, it is easy to find a proper tree with the same consistent set. If T and T' are proper, it may be shown that cs(T) = cs(T') holds if and only if T and T' are equivalent. Now suppose T is the PQ-tree constructed for the graph G. Place T into proper form. Let the characteristic node of a set S be the node of minimum height whose frontier contains S, where the height of a node t is the maximum distance from t to a leaf in the frontier of t. Associate each vertex of G with the characteristic node of C(v). It is possible to assign a label to each node t which gives sufficient information about the sets C(v), for all vertices associated with t, that the graph could be reconstructed. An algorithm similar to Example 3.2 of [AHU] may then be used to decide whether two such labelled trees are equivalent. This gives the following.

Theorem 4.5. Interval graph isomorphism may be decided in linear time.

This is particularly interesting since for two of the best known classes of graphs which contain the interval graphs, namely the chordal graphs [LB] and the graphs whose complement is transitively orientable [GH], isomorphism is not much easier than for arbitrary graphs, as we can show.

Theorem 4.6. Arbitrary graph isomorphism is polynomially reducible to chordal graph isomorphism, and to transitively orientable graph isomorphism. (Note that the latter implies that arbitrary graph isomorphism is polynomially reducible to isomorphism of graphs whose complement is transitively orientable. For more information about the notion of polynomial reducibility, see [Ka].)

Proof sketch. We construct a polynomial mapping M from a graph G to a graph M(G) such that M(G) is chordal (respectively transitively orientable), and G can be recovered from M(G) up to isomorphism. Then the question of whether G_1 is isomorphic to G_2 is reduced to the question of whether $M(G_1)$ is isomorphic to $M(G_2)$.

For the reduction to chordal graph isomorphism, let M(G) = (V',E'), where

$$V' = V \cup E, \text{ and}$$

$$E' = \{\{v,w\} \mid v \neq w \text{ and } v,w \in V\} \cup \{\{v,u\} \mid v \in V,\ u \in E, \text{ and } v \text{ is incident to } u\}.$$

M(G) is readily seen to be chordal, and if G has at least four vertices, we can recover G from M(G) up to isomorphism.

For the reduction to transitively orientable graph isomorphism, create two new vertices x and y, and let

$$V' = V \cup E \cup \{x,y\}, \text{ and}$$

$$E' = \{\{v,u\} \mid v \in V,\ u \in E, \text{ and } v \text{ is incident to } u\} \cup \{\{y,v\} \mid v \in V\} \cup \{\{x,y\}\}.$$

M(G) is transitively orientable. Also, G can be reconstructed from M(G) up to isomorphism. □

5. Conclusion.

This paper gives a variety of applications of a data structure called a PQ-tree; it also presents a new application of lexicographic breadth first search. These applications include linear algorithms to recognize interval graphs, test for the consecutive ones property, and perform a number of related tasks. The results are presented here in capsule form; later papers will give more details.

Acknowledgement.

The authors are very grateful to Richard Karp, Robert Tarjan, and Jeffrey Ullman for their many helpful comments and suggestions.

References.

[AHU] Aho, A.V., Hopcroft, J.E., and Ullman, J.D., The Design and Analysis of Computer Algorithms, Addison-Wesley, Reading, Mass., 1974.

[B] Benzer, S., "On the Topology of the Genetic Fine Structure," Proc. Nat. Acad. Sci. U.S.A., 45 (1959), pp. 1607-1620.

[BT] Booth, K.S., and Tarjan, R.E., "Layered Breadth-First Search and Chordal Graphs," draft.

[C] Cohen, J., "Interval Graphs and Food Webs," The RAND Corporation, D-17696-PR.

[CG] Coffman, E.G., Jr., and Graham, R.L., "Optimal Scheduling for Two-Processor Systems," Acta Informatica, 1 (1972), pp. 200-213.

[E] Even, S., private communication to R. Tarjan.

[FG] Fulkerson, D.R., and Gross, O.A., "Incidence Matrices and Interval Graphs," Pac. J. Math., 15 (1965), pp. 835-855.

[G] Ghosh, S.P., "On the Theory of Consecutive Storage of Relevant Records," Information Sciences, 6 (1973), pp. 1-9.

[GH] Gilmore, P.C., and Hoffman, A.J., "A Characterization of Comparability Graphs and of Interval Graphs," Can. J. Math., 16 (1964), pp. 539-548.

[H] Hajos, G., "Über eine Art von Graphen," Internationale Math. Nachrichten, 11 (1957), p. 65.

[HT] Hopcroft, J.E., and Tarjan, R.E., "Efficient Planarity Testing," J. ACM, 21 (1974), pp. 549-568.

[Ka] Karp, R.M., "Reducibility among Combinatorial Problems," in Complexity of Computer Computations, R. E. Miller and J.W. Thatcher, eds., Plenum Press, New York, 1972, pp. 85-103.

[Ke] Kendall, D.G., "Some Problems and Methods in Statistical Archaeology," World Archaeology, 1 (1969), pp. 68-76.

[L] Lueker, G.S., "Structured Breadth First Search and Chordal Graphs," Technical Report 158, Dept. of Elec. Eng., Computer Science Lab., Princeton University, August 1974.

[LB] Lekkerkerker, C.G., and Boland, J. Ch., "Representation of a Finite Graph by a Set of Intervals on the Real Line," Fund. Math., 51 (1962), pp. 45-64.

[LEC] Lempel, A., Even, S., and Cederbaum, I., "An Algorithm for Planarity Testing of Graphs," Theory of Graphs: International Symposium: Rome, July, 1966, P. Rosenstiehl, ed., Gordon and Breach, New York, 1967, pp. 215-232.

[PLE] Pnueli, A., Lempel, A., and Even, S., "Transitive Orientation of Graphs and Identification of Permutation Graphs," Can. J. Math., 23 (1971), pp. 160-175.

[R] Roberts, F.S., Discrete Mathematical Models, with Applications to Social, Biological and Environmental Problems, Prentice-Hall, Englewood Cliffs, N.J., to appear.

[R68] Roberts, F.S., Representations of Indifference Relations, Ph.D. Thesis, Stanford University, 1968.

[R69] Roberts, F.S., "Indifference Graphs," in Proof Techniques in Graph Theory, F. Harary, ed., Academic Press, New York, 1969, pp. 139-146.

[RTL] Rose, D.J., Tarjan, R.E., and Lueker, G.S., "Algorithmic Aspects of Vertex Elimination on Graphs," draft.

[Se] Sethi, R., "Algorithms for Nonpreemptive Scheduling on Two Processors," Computer Science Department, Pennsylvania State University, unpublished manuscript, 1974.

[St] Stoffers, K.E., "Scheduling of Traffic Lights--a New Approach," Transportation Research, 2 (1968), pp. 199-234.

[Ta] Tarjan, R.E., "Implementation of an Efficient Algorithm for Planarity Testing of Graphs," Dec. 1969, unpublished manuscript.

[T71] Tucker, A., "Matrix Characterizations of Circular-Arc Graphs," Pac. J. Math., 39 (1971), pp. 535-545.

[T72] Tucker, A., "A Structure Theorem for the Consecutive 1's Property," J. Comb. Theory, 12(B) (1972), pp. 153-162.

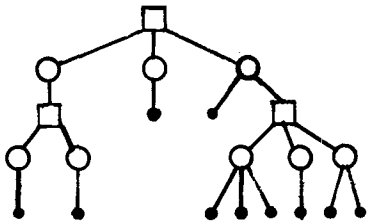


Figure 2.1. A PQ-tree.

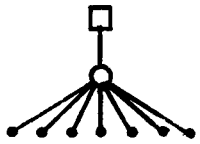


Figure 2.2. A universal PQ-tree.

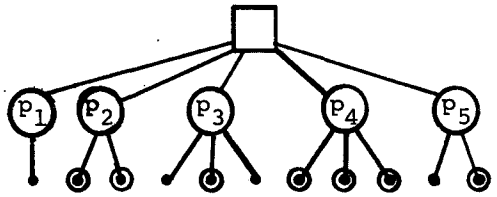


Figure 2.3. Illustration of the function SCAN. SCAN(p_3) returns the value p_2 or p_5. In this and subsequent figures, elements of S are circled.

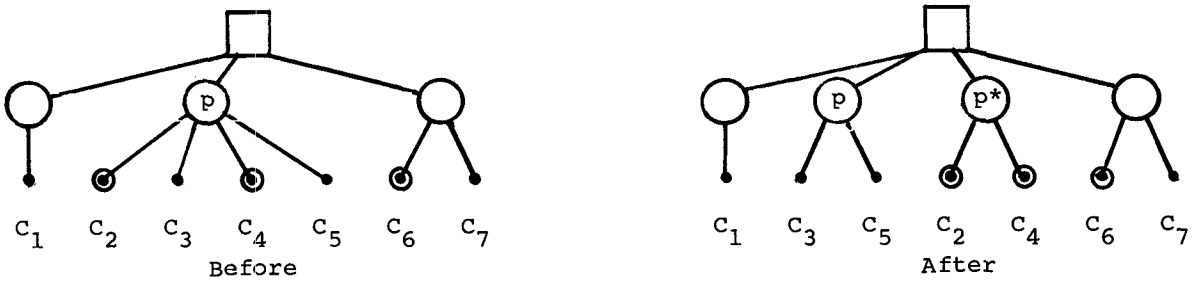


Figure 2.4. Example of the effect of SPLIT(p,p*).

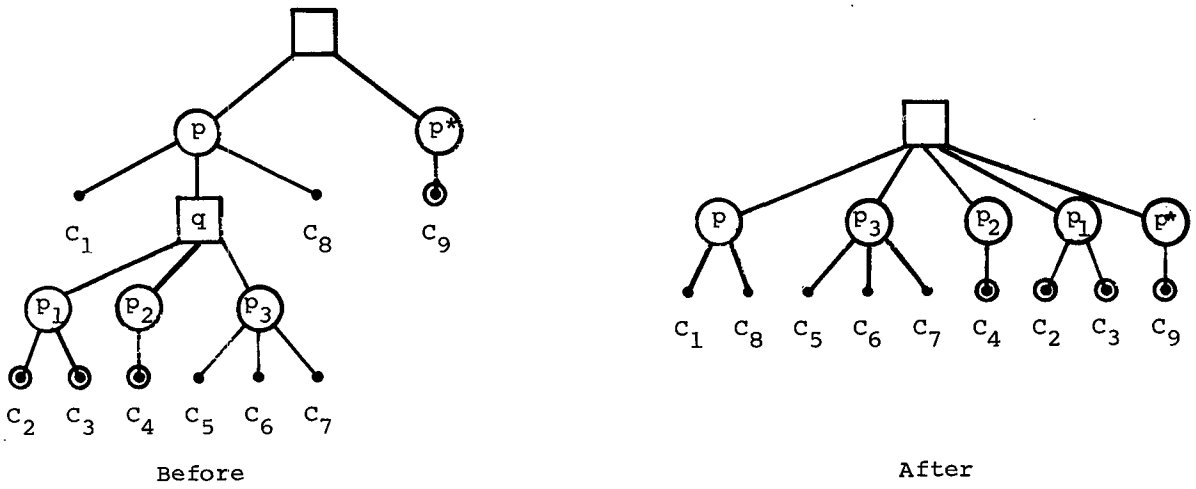


Figure 2.5. Example of the effect of COLLAPSE(p,q,p*).

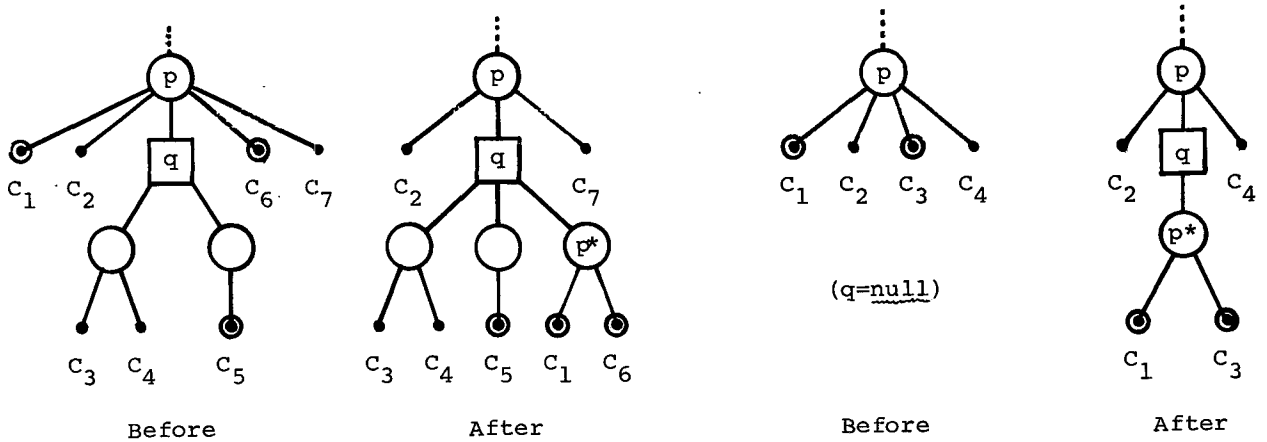
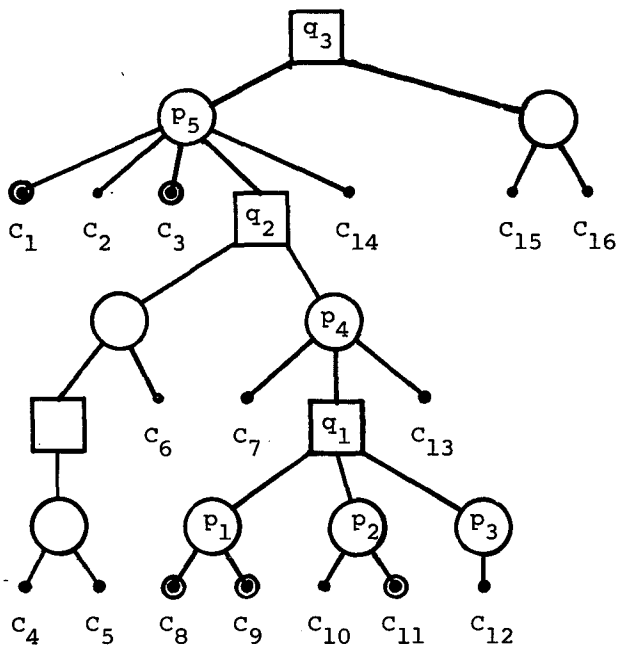
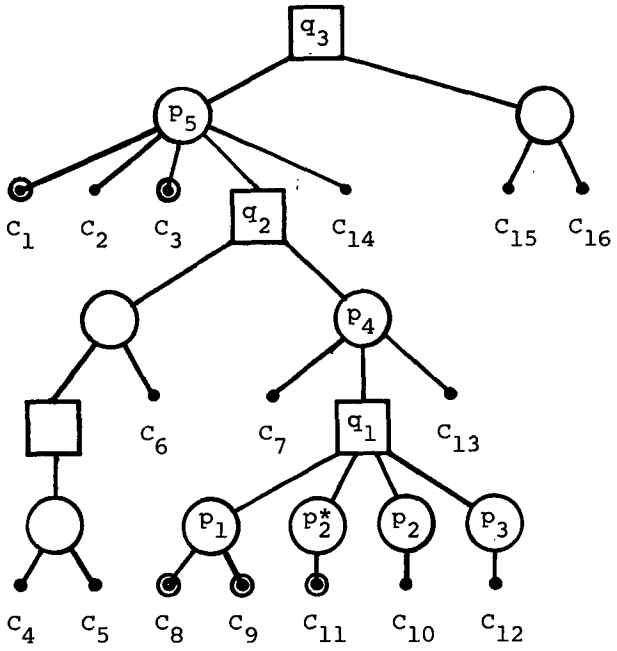


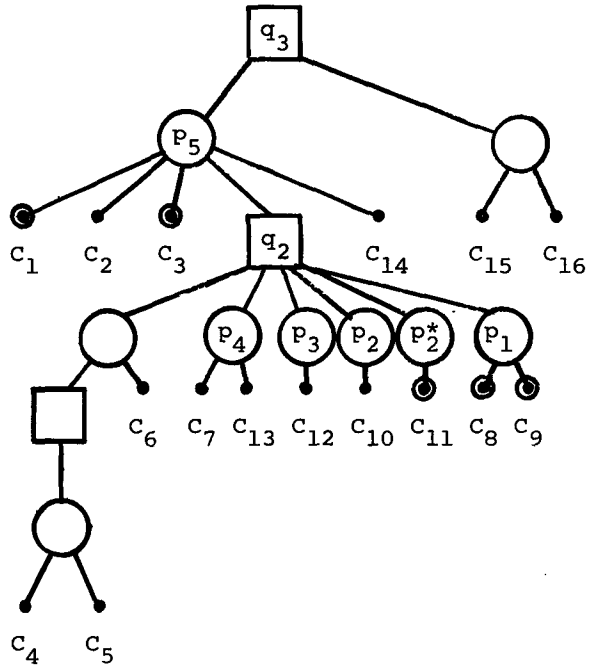
Figure 2.6. Two examples of the effect of EXTEND(p,q,p*).



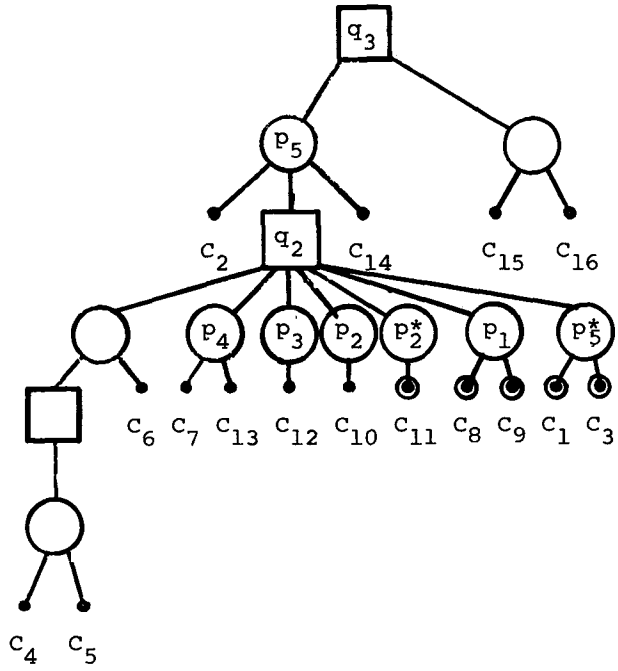
a) Before the call to REDUCE.



b) After one pass through LOOP.



c) After two passes through LOOP.



d) At the end of REDUCE.

Figure 2.7. Illustration of REDUCE(S). A tree is shown at various times during the execution of the routine.