UC Irvine
ICS Technical Reports

Title
A linear time algorithm for deciding interval graph isomorphism

Permalink
https://escholarship.org/uc/item/8ss2q7bp

Authors
Lueker, George S.
Booth, Kellogg S.

Publication Date
1977

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

A LINEAR TIME ALGORITHM FOR DECIDING INTERVAL GRAPH ISOMORPHISM

by

George S. Lueker
Kellogg S. Booth

Notice: This Material may be protected by Copyright Law (Title 17 U.S.C.)

Technical Report #109

Department of Information and Computer Science
University of California, Irvine
Irvine, California
92717

A LINEAR TIME ALGORITHM FOR DECIDING INTERVAL GRAPH ISOMORPHISM

by

George S. Lueker+

and

Kellogg S. Booth++

Z
699
C3
no. 109

+Supported by the National Science Foundation under Grant MCS77-04410 at Irvine and by NSF Grant GJ-1052 at Princeton University

++Supported by the National Research Council of Canada under Grant No. A-4307 at Waterloo and by the U.S. Energy Research and Development Agency under Contract No. W-7405-Eng-48 at Lawrence Livermore Laboratory

Authors' addresses:

George S. Lueker
Department of Information and Computer Science
University of California, Irvine
Irvine, California 92717

Kellogg S. Booth
Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada N2L 3G1

Abstract

A graph is an interval graph if and only if each of its vertices can be associated with an interval on the real line in such a way that two vertices are adjacent in the graph exactly when the corresponding intervals have a nonempty intersection. An efficient algorithm for testing isomorphism of interval graphs is implemented using a data structure called a PQ-tree. The algorithm runs in $O(n + e)$ steps for graphs having n vertices and e edges. It is shown that for a somewhat larger class of graphs, namely the chordal graphs, isomorphism is as hard as for general graphs.

1. Introduction.

Let G be a graph; let V be its set of vertices and let E be its set of edges. Let n be the number of vertices and e be the number of edges. G is said to be the intersection graph of a family of sets $F = \{S_i\}_{i=1}^{n}$ if there is a one-to-one correspondence between V and F such two vertices are adjacent if and only if the corresponding sets have a nonempty intersection. For example, G is said to be an interval graph if it is the intersection graph of a family of intervals on the real line, that is, if it is possible to set up a one-to-one 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 is called an intersection model for G. Figure 1 gives an example of an interval graph and its intersection model. The problem of characterizing such graphs was posed by Hajós [8]. Since then a number of interesting applications for interval graphs have been found; [15] contains a survey of these applications. A somewhat larger class of graphs is the class of chordal graphs. If C is a cycle in a graph G, C is said to have a chord if there is an edge in G, other than the edges of the cycle, which connects two vertices of the cycle. A graph is a chordal graph if every cycle in G of length greater than three has a chord. In addition to being called chordal graphs [7], such graphs have also been called triangulated graphs [16], rigid circuit graphs [6], and acyclic graphs [12]. It is known that

any interval graph is also a chordal graph [12]; in fact, it has been shown that a graph is a chordal graph if and only if it is the intersection graph of a family of subtrees in a tree [7].

Although the general problem of determining graph isomorphism appears to be hard, for some special classes of graphs isomorphism can be decided efficiently. For example, for planar graphs, it can be decided in O(n + e) time [10]. In this paper we discuss the problem of deciding isomorphism for interval graphs and for chordal graphs. We show that for interval graphs, isomorphism can be decided in O(n + e) time; for chordal graphs, on the other hand, we show it to be as hard as arbitrary graph isomorphism. Most of the ideas in this paper can be found in the authors' theses [2, 14]. Additional results, dealing with the complexity of isomorphism for a class of graphs between the interval graphs and chordal graphs, can be found in [3].

2. A characterization of interval graph isomorphism.

A clique of a graph G is a maximal subset of V all of whose elements are adjacent. For each vertex v, let C(v) be the set of cliques which contain v. The following characterization is due to Fulkerson and Gross.

Theorem [6]. G is an interval graph if and only if there exists a linear ordering of its cliques such that for each vertex v, the elements of C(v) appear consecutively within the ordering.

In the interval graph recognition algorithm in [6], G is immediately rejected if it fails to be a chordal graph; if G is chordal, [6] demonstrates a polynomial time algorithm to construct all of its cliques. It is easy to see from this algorithm that a chordal graph has at most n cliques, and that the sum of their sizes is O(n + e).

The Fulkerson-Gross characterization is used in [4] to produce the following algorithm to test whether G is an interval graph.

Algorithm 1. Interval graph recognition.

```
boolean procedure INTERVAL(G);
    begin
       if G is not a chordal graph then return false;
       let Π be the set of all orderings of the cliques
        of G;
       for each vertex v ∈ V do
          remove from Π those permutations which do not
           have all elements of C(v) appearing consecutively;
       if Π ≠ ∅
          then return true
          else return false
    end.
```

Recently O(n + e) algorithms have been developed to determine whether G is chordal and to find all of its cliques [14, 16]; nonetheless, Algorithm 1 at first appears to be terribly inefficient, since the set Π which is being manipulated can be very large. However, [4] shows how to implement this algorithm to run in O(n + e) time by a very careful choice of the data structure for Π; this data structure is called a PQ-tree. A PQ-tree is an ordered tree whose nonterminal nodes (i.e., all nodes except the leaves) fall into two classes, namely the P-nodes and Q-nodes; a nonterminal node is said to be of type P or Q. Two trees T and T′ are said to be equivalent, written $T = T'$, 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 ordering of the children of a Q-node.

(A structure similar in principle to a PQ-tree was used in a planar graph recognition algorithm [13], although it was not described as a tree; in [4] it is shown how the idea in [13] may be streamlined and implemented efficiently to produce several efficient algorithms.)

A PQ-tree is proper if each P-node has at least 2 children, and each Q-node has at least 3 children. The frontier of a PQ-tree is the ordering of its leaves obtained by reading them from left to right. The frontier of a node t, written F(t), is the frontier of the subtree rooted at t. An ordering of the leaves of T is consistent with T if it is the frontier of a tree equivalent to T. The set of all orderings consistent with T is denoted CONSISTENT(T). Figure 2 shows a PQ-tree and two other equivalent PQ-trees; in this and subsequent figures, P-nodes are drawn as circles, Q-nodes as rectangles, and leaves as dots. When it is desirable to represent a node without specifying its type, a small triangle will be used; large triangles will represent subtrees.

In our applications, we will let the leaves of T be the the cliques of G. It turns out that Algorithm 1 can be implemented efficiently by manipulating a tree T such that TT is CONSISTENT(T). In fact, the following may be proven.

Theorem [4]. Interval graph recognition may be performed in $O(n + e)$ time. Moreover, if G is an interval graph, then there is an $O(n + e)$-time algorithm to construct a proper PQ-tree T such that CONSISTENT(T) is precisely the set of orderings of the cliques of G in which C(v) appears consecutively for all v.

An implementation of this test has been programmed [17]; the implementation contains a number of interesting ideas not present in [4]. Also, Fischer and Ladner have made numerous helpful suggestions about the implementation of the algorithm [5].

The tree guaranteed by this theorem will henceforth be denoted T(G). We begin to attack the question of determining isomorphism of two interval graphs G and G' by comparing T(G) and T(G').

The following theorem tells us that isomorphic graphs will have equivalent PQ-trees.

Theorem 1: If T_1 and T_2 are PQ-trees, with the same set of leaves, such that CONSISTENT(T_1) = CONSISTENT(T_2), then $T_1 \equiv T_2$.

Proof: If two trees have different numbers of leaves, they can clearly neither be equivalent nor have identical consistent sets. Thus we may assume both trees have the same number of leaves; let this common number be m. We now prove the theorem by induction on m. If m=1 then there is only a

single PQ-tree, consisting of only one leaf node, and the theorem is trivially true.

Assume then that the theorem is true for all trees with less than m leaves, where m > 1. Let T_1 and T_2 be two trees having m leaves with CONSISTENT(T_1) = CONSISTENT(T_2). We claim that the roots of T_1 and T_2 (necessarily nonterminal nodes) either are both P-nodes having subtrees which can be corresponded so that pairwise they are isomorphic or else they are both Q-nodes whose subtrees in left-to-right order (or the reversal) are pairwise isomorphic. This is sufficient to establish that $T_1 \equiv T_2$.

We prove the claim by first showing that each root has the same number of subtrees, and that these subtrees partition the leaves in the same way; that is, we show that there is a one-to-one correspondence between the subtrees of T_1 and T_2 such that corresponding trees have the same sets of leaves. Pick any subtree T' of the root of T_1. Its leaves appear somewhere in T_2.

Suppose these leaves occur in more than one subtree of the root of T_2. Consider two such subtrees. If either one of these subtrees does not consist entirely of leaves from T' then we have the situation on the left in Figure 3, where x and y are leaves in one subtree of T_2, z is in a different subtree of T_2, and both y and z are in T' but x is not. We can reverse the entire left subtree in T_2 to obtain an equivalent tree in which x separates y and z; this is shown

on the right of Figure 3. This is a contradiction because y and z can only be separated by leaves of T'. Thus we know that if the leaves of T' occur in more than one subtree of T_2 then they comprise all of the leaves of those subtrees.

Continuing the assumption that the leaves of T' are in more than one subtree of T_2, let y and z be two leaves of T' which are in distinct subtrees of T_2; note that since T_1 has at least one other subtree in addition to T' we can choose a leaf x which is not in T'. By the discussion above we know that x is in no subtree of T_2 which contains leaves from T' and thus x, y, and z appear in distinct subtrees in T_2. This implies that the root of T_2 is a Q-node since otherwise we could rearrange three of its subtrees as in Figure 4 and obtain a contradiction as before. But if the root is a Q-node we are again in trouble. We can rearrange T_1 so that x is closer to either y or z; see Figure 5. This contradicts the fact that if x, y, and z are in separate subtrees of the root of T_2, and if this root is a Q-node, then the relative order of x, y, and z is determined up to reversal. We are thus left with the conclusion that all leaves of T' must occur within a single subtree of T_2. Thus the partition of the leaves of T_1 induced by the subtrees of its root must be a refinement of the partition of the leaves of T_2 induced by the subtrees of its root. A symmetric argument shows that the same statement is true if we interchange the roles of T_1 and T_2. Thus the two partitions must in fact be identical, as was desired. Note that this implies that both roots have the same number of

children; call this number r.

The next claim is that the roots are either both P-nodes or both Q-nodes. If r = 2 both roots must be P-nodes, since the trees are proper. If $r \geq 3$, and the roots are of different types, assume without loss of generality that T_1 has a P-node root and T_2 has a Q-node root. Then we could choose leaves x, y, and z such that we have the situation in Figure 6 in which y always occurs between x and z in T_2 but not in T_1. This verifies the second claim.

Finally, we see that the subtrees are pairwise isomorphic (given the pairing determined by the partitioning of leaves) by use of the inductive hypothesis. []

Theorem 1 enables us to conclude that if G and G´ are isomorphic, then T(G) and T(G´) are equivalent. Now we attack the converse problem: we wish to guarantee that if T(G) and T(G´) are equivalent, then G and G´ are isomorphic. Unfortunately, however, this is not the case; it is possible for graphs which are not isomorphic to have equivalent PQ-trees. Figure 7 illustrates the problem we face. In order to overcome this problem, we will have to modify the PQ-tree so that it gives more information about the structure of the graph. For any vertex v in G, let the characteristic node of v, written CHAR(v), be the deepest node t in T such that F(t) contains C(v). We will often use the inverse image of this function; if t is a node in T, let

$$CHAR^{-1}(t) = \{v \in V | t = CHAR(v)\}.$$

Given a subset S of the leaves of T, it is useful to classify a node t of T as pertinent with respect to S if F(t) contains some element of S and empty with respect to S if F(t) does not contain any element of S; the pertinent nodes are further classified as full with respect to S if F(t) is composed entirely of elements of S, or partial with respect to S if F(t) contains some elements in S and some not in S. When the set S is clear from the context, the phrase "with respect to S" will be omitted.

Lemma 1. Let S be a nonempty set which appears consecutively in all elements of CONSISTENT(T). Then either

a) there is a P-node or leaf whose frontier comprises precisely S, or

b) there is a consecutive sequence of children of a Q-node q such that the union of the frontiers of the nodes in the sequence is S.

Proof. Let t be the deepest node in T whose frontier contains S. Consider three cases.

Case 1. Node t is a leaf. Then the frontier of t contains only t, so (a) holds trivially.

Case 2. Node t is a P-node. We will show that F(t) comprises precisely S. Assume the contrary in order to derive a contradiction. Since t is the deepest node whose frontier contains S, t must have at least two pertinent children; also, by assumption, t must have at least one partial or empty child. Thus, we may consider two subcases.

Subcase 2.1. Node t has an empty child. Permute the children of t so that the empty child appears between two pertinent children. Then clearly S is not consecutive in the frontier; this contradicts the hypothesis of the theorem.

Subcase 2.2. Node t has a partial child t_1. Let C_f be a leaf which descends from t_1 and is in S; similarly, let C_e be a leaf which descends from t_1 and is not in S. Recall that t has at least two pertinent children, and let C_f' be a descendant in S of a pertinent child of t other than t_1. If C_e appears between C_f and C_f', S is clearly not consecutive in the frontier, a contradiction. Otherwise, reverse the frontier of t_1 by reversing the children of all nodes which descend from t_1 (including t_1 itself). Then C_e will appear between C_f and C_f', so we again have the desired contradiction.

Case 3. Node t is a Q-node. Let $t_1, t_2, \ldots, t_k$ be the children of t in order from left to right. Let t_i (respectively t_j) be the leftmost (respectively rightmost) pertinent child of t. Note that since t is the deepest node whose frontier contains S, i must not equal j. Now all

children between t_i and t_j must be full, since otherwise S would not be consecutive in the frontier. Thus all that remains to be shown is that t_i and t_j are full. Assume for a contradiction that they are not both full; without loss of generality assume that t_i is partial. Then as in Subcase 2.2, either S is not consecutive in the frontier, or by reversing the frontier of t_i we can prevent S from being consecutive. This final contradiction completes the proof. []

We will now attach labels to the nodes of the PQ-trees; these labels will consist of strings of integers which indicate how the sets C(v) are distributed over the frontier of tne tree. The labels are defined as follows.

a) If t is a P-node or a leaf, LABEL[t] is set to $|CHAR^{-1}(t)|$, i.e., the number of vertices of G which have t as their characteristic node.

b) If t is a Q-node, number the children of t as $t_1, t_2, \ldots, t_k$ from left to right. For each v in $CHAR^{-1}(t)$ form a pair (i,j) such that t_i is the leftmost child of t which is pertinent and t_j is the rightmost. Sort all of these pairs into lexicographically nondecreasing order and concatenate them to form LABEL[t].

The resultant labeled tree is denoted $T_L(G)$. This labeling is illustrated in Figure 8.

Theorem 2. A labeled PQ-tree contains enough information to reconstruct G up to isomorphism.

Proof. Given a labeled PQ-tree T, we construct a graph G′ on a set V′ of vertices which contains the following elements.

a) For each leaf or P-node t with label m, there are m vertices, namely, (t,r), for $1 \leq r \leq m$; these correspond to the m elements of $CHAR^{-1}(t)$.

b) For each Q-node q with label

$$LABEL[q] = (i_1,j_1),(i_2,j_2),\ldots,(i_m,j_m),$$

there are m vertices, namely the quadruples

(q,r,i_r,j_r), for $1 \leq r \leq m$.

In view of the definition of the labels, these correspond in a natural way to elements of $CHAR^{-1}(q)$.

Associate with each vertex $v' \in V'$ a set $C'(v')$ as follows. If v′ has the form (t,r), let $C'(v')$ be the set of elements of F(t). If v′ has the form (q,r,i,j), let $C'(v')$ be the total

set of elements of the frontiers of the i^{th} through j^{th} child of q. By Lemma 1 and the definitions of the labels, if v and v′ are corresponding elements of V and V′, then C(v)=C′(v′). Form G′ by letting v′ and w′ be adjacent if and only if C′(v′) and C′(w′) intersect. Then since vertices of G are adjacent if and only if they are contained in a common clique, G and G′ are isomorphic. []

In order to describe the test for isomorphism, we define the following relations. Two labeled PQ-trees T and T′ are L-identical, written $T=_L T'$, if they are isomorphic as ordered trees, and corresponding nodes have equal labels. Two labeled PQ-trees T and T′ are L-equivalent, written $T \equiv_L T'$, if T can be made L-identical to T′ by any sequence of equivalence transformations, providing we always appropriately modify labels of Q-nodes whose children are reversed. It is not hard to see that the required change for a Q-node with k children can be performed by

a) replacing each pair (i,j) in LABEL[q] by the pair (k+1-j,k+1-i), and

b) re-sorting the pairs into lexicographically nondecreasing order.

Theorem 3. Two graphs G and G´ are isomorphic if and only if $T_L(G) = T_L(G')$.

Proof. This follows easily from Theorem 1 and Theorem 2. []

3. The interval graph isomorphism algorithm.

Let N be the number of nodes in the PQ-tree under consideration. A method for labeling a PQ-tree is shown in Algorithm 2. It is assumed that each node has a field CCOUNT which tells the number of children. Moreover, each node has a field NUMBER; if the sequence of children of a node, from left to right, is $t_1, t_2, \ldots, t_k$, then NUMBER[t_i] is i. It is easy to number the nodes in O(N) time.

Algorithm 2. Labeling a PQ-tree.

```
procedure LABEL(T);
  begin
    for each node t in T
      begin
        if t is a Q-node
          then LABEL[t] <- the empty string;
          else LABEL[t] <- ∅;
        comment FCOUNT[t] will contain the number
          of full children of t;
        FCOUNT[t] <- ∅;
        comment SCOUNT[t] will contain the number
          of descendants of t which are in C(v);
        SCOUNT[t] <- ∅;
      end;

    VLOOP: for each vertex v in V do
      begin
        QUEUE <- a list of all elements of C(v);
        comment nodes found to be full will be
          added to QUEUE;
        FLIST <- a list of all elements of C(v);
        for all leaves C in C(v) do
          SCOUNT[C] <- 1;
        FLOOP: while QUEUE is not empty do
          begin
            remove an element t from the head of QUEUE;
            if SCOUNT[t] ≠ |C(v)| then
              begin
                t´ <- PARENT[t];
                FCOUNT[t´] <- FCOUNT[t´] + 1;
                SCOUNT[t´] <- SCOUNT[t´] + SCOUNT[t];
                if FCOUNT[t´] = CCOUNT[t´] then
                  begin
                    comment t´ is full;
                    append t´ to the end of QUEUE;
                    append t´ to FLIST;
                    for all children t" of t´ do
                      remove t" from FLIST;
                  end;
              end;
          end FLOOP;
```

```
      t <- any element of FLIST;
      if |FLIST| = 1
        then if t is not a Q-node
          then LABEL[t] <- LABEL[t] + 1
          else append the pair (1,CCOUNT[t]) to
            LABEL[t]
        else
          begin
            q <- PARENT[t];
            LEFT <- CCOUNT[q];
            RIGHT <- 0;
            for each element t of FLIST do
              begin
                LEFT <- min(LEFT,NUMBER[t]);
                RIGHT <- max(RIGHT,NUMBER[t]);
              end;
            append the pair (LEFT,RIGHT) to LABEL[q];
          end;
      RESET: reset all the modified FCOUNT and SCOUNT
        fields to zero;
    end VLOOP;
  SORT: for each Q-node q do
    sort the pairs in LABEL[q] into lexicographically
      nondecreasing order;
end.
```

Lemma 2. Algorithm 2 is correct and can be implemented to run in O(n + e) time.

Proof. We begin the proof of correctness by showing that during each pass through VLOOP,

a) if CHAR(v) is a P-node or leaf, say t, then LABEL[t] is incremented, while

b) if CHAR(v) is a Q-node q, the pair (LEFT,RIGHT) is appended to LABEL[q], where LEFT (respectively RIGHT) is the NUMBER of the leftmost (respectively rightmost) pertinent child of q.

c) LABEL[x] is unchanged for other nodes x.

In the algorithm, FCOUNT[t] gives the number of full children of t, and SCOUNT[t] gives the number of descendents in C(v) of t. QUEUE contains nodes which are found to be full. At each iteration of FLOOP, a full node t is removed from QUEUE. If SCOUNT[t] $\neq$ |C(v)|, i.e., if F(t) does not contain C(v), then the field FCOUNT of the parent t' of t is incremented. If FCOUNT[t'] becomes equal to CCOUNT[t'], i.e., if t' is found to be full, then t' is added to the queue. By a simple induction on the level of nodes, we may conclude that the nodes added to FLIST are precisely the full nodes in the tree. However, a node is deleted when its parent is added, so we conclude that at the end of FLOOP, FLIST contains a node t if and only if t is full but the parent of t is not.

Now let t=CHAR(v) and consider two cases.

Case 1. Node t is full. Then by the above remarks, at the end of FLOOP the only node remaining on FLIST is t. Looking at the algorithm, we see that LABEL[t] is modified accordingly to its definition: if t is a P-node or leaf, its LABEL is incremented; if t is a Q-node, the pair (1,CCOUNT[t]) is added to LABEL[t], since all children of t are full.

Case 2. Node t is not full. Then by Lemma 1 and the above remarks, the nodes on FLIST must form a consecutive sequence of siblings whose frontiers together comprise precisely C(v); moreover, their parent must be a Q-node q and must be the characteristic node of v. Then in the algorithm, LEFT and RIGHT become the numbers of the leftmost and rightmost elements of this sequence, and the pair (LEFT,RIGHT) is added to LABEL[q], as desired.

We now know that at the termination of VLOOP, LABEL[t] is correct for all P-nodes and leaves; moreover, for all Q-nodes q, LABEL[q] contains the correct pairs, possibly in the wrong order. Thus after the lexicographic sorts performed in step SORT, the tree is correctly labeled.

To see that the algorithm is linear, first consider the time spent in VLOOP. Note that since T is proper each node has at least two children. From this it follows readily that the number of full nodes with respect to a set S is linear in

|S|. Thus the number of passes through FLOOP for any vertex is O(|C(v)|); summing this over all vertices gives O(n + e). At first glance, step RESET may seem to require time proportional to the number of nodes in the tree; however, if we maintain a list of nodes whose fields are changed, we may reset their count fields to ∅ in O(|C(v)|) time. The remainder of the processing for a vertex is also easily seen to be O(|C(v)|), so the amount of time spent in VLOOP is linear. Now consider the time spent in the sorts in step SORT. We use Algorithm 3.1 of [1], so the time for one sort is bounded by the length of LABEL[q] plus the range of values of the elements of LABEL[q]. From the definition of the labels, a bound is

$$O(|CHAR^{-1}(q)| + CCOUNT[q]). \qquad (1)$$

Summing (1) over all Q-nodes gives O(n + N), which is O(n).

[]

To test for L-equivalence of trees, we may use the following algorithm, which is based on the tree isomorphism algorithm in [1]. The algorithm places a labeled tree T into a form which is canonical for L-equivalence; that is, T and T′ are L-equivalent if and only if the algorithm maps them into the same tree. A bit of explanation of the notation used in the algorithm will be helpful. $LABEL^r(q)$ denotes the label q would have if its children were reversed. A[t] is a sequence which is associated with a node t during the

algorithm: it consists of a string over the positive integers and the symbols 'C', 'P', and 'Q'. The index of a sequence in a family of sequences is defined as follows. Given a family of sequences $A_1, A_2, \ldots, A_k$, sort them into lexicographically nondecreasing order. Eliminate duplicates. If a sequence A is the i^{th} element in the resulting ordering of distinct sequences, we say that i is the index of A in the family. When performing a sort, we shall arbitrarily adopt the collating sequence

$$'C' < 'P' < 'Q' < 1 < 2 < \ldots .$$

Two consecutive vertical bars (||) will be used to indicate concatenation of sequences.

Algorithm 3. Transformation of a labeled PQ-tree into canonical form for L-equivalence.

```
procedure CANONICAL(T);
    begin
     for m <- 0 to height(T) do
       LOOP: begin
         for each leaf C at level m do
           Cloop: A[C] <- LABEL[C] || 'C';
         for each P-node p at level m do
           Ploop: begin
             rearrange the children of p into an order
              t_1,t_2,...,t_k so that the sequence
              a[t_1],a[t_2],...,a[t_k] is nondecreasing:
             A[p] <- LABEL[p] || 'P' ||
              a[t_1],a[t_2],...,a[t_k];
           end;
         for each Q-node q at level m do
           Qloop: begin comment let t_1,t_2,...,t_k be the
                  children of q, in order from left to right;
             L1 <- LABEL[q];
             L2 <- LABEL^r[q];
             A1 <- L1 || 'Q' || a[t_1],a[t_2],...,a[t_k];
             A2 <- L2 || 'Q' || a[t_k],a[t_k-1],...,a[t_1];
             if A1 ≤ A2
               then A[q] <- A1
               else
                 begin
                   reverse the order of the children of q;
                   A[q] <- A2;
                   LABEL[q] <- L2;
                 end;
           end;
         INDICES: 𝔸_m <- {A[t]|t is at level m};
         for each t at level m do
           a[t] <- the index of A[t] in 𝔸_m;
       end;
    end.
```

Lemma 3. Algorithm 3 correctly places a labeled PQ-tree into canonical form. That is, if T_1 and T_2 are L-equivalent, and the algorithm transforms them into T_1' and T_2', respectively, then T_1' and T_2' are L-identical.

Proof. Suppose we run the algorithm once on T_1 and once on T_2. Let i and j each be 1 or 2; i and j may or may not be equal. Let t (respectively t') be a node at level m in T_i (respectively T_j). If t_a is a node in a tree T_a, define $T_a(t_a)$ to be the subtree in T_a which is rooted at t_a; that is, $T_a(t_a)$ is the tree which includes t_a and all of its descendants. We prove by induction on m that the following four statements are equivalent:

a) $T_i(t) =_L T_j(t')$

b) $A[t] = A[t']$

c) $T_i'(t) =_L T_j'(t')$

d) $a[t] = a[t']$

Note that the lemma will then follow immediately from (a) and (c) if we let t (respectively t') be the root of T_1 (respectively T_2).

Basis (m=∅). Then t and t' are leaves. Thus (a) and (c) are trivially equivalent; they are also equivalent to (b) from step Cloop of the algorithm. If i=j, (b) <==> (d) follows directly from the fact that the index function is one-to-one. If i≠j, note that the set $\mathbb{A}_\emptyset$ found in the algorithm when executed on T_1 is the same as the set $\mathbb{A}_\emptyset$ found when the algorithm is executed on T_2. This follows from the

fact that if two trees are L-equivalent, the set of subtrees of level m in one tree may be put into a one-to-one correspondence with the set of subtrees at level m in the other tree in such a way that corresponding subtrees are L-equivalent; then letting m=∅ and using the fact that (a) <==> (b), we see that $\mathbb{A}_{\emptyset}$ must be the same for T_1 and T_2. Thus again (b) <==> (d) follows from the fact that the index function is one-to-one.

Inductive step. Assume that t and t′ are a level m and we know that (a) through (d) are equivalent for nodes at lower levels. We begin the proof of the inductive step by showing that (a) ==> (b) for t and t′. Note that t and t′ have the same type since they are roots of L-equivalent subtrees. Consider three cases.

Case 1. Nodes t and t′ are leaves. Then the argument of the basis holds.

Case 2. Nodes t and t′ are P-nodes. Since $T_i(t) \equiv_L T_j(t')$, we must have LABEL[t]=LABEL[t′]. Let the children of t (respectively t′) be $\{t_r\}_{r=1}^{k}$ (respectively $\{t'_r\}_{r=1}^{k}$). Again since $T_i(t) \equiv_L T_j(t')$, it must be that there exists a one-to-one correspondence between the t_r and t'_r such that subtrees rooted at corresponding nodes are L-equivalent. Thus by the inductive hypothesis the two multisets $\{a[t_r]\}_{r=1}^{k}$ and $\{a[t'_r]\}_{r=1}^{k}$ are the same. Therefore block Ploop in the algorithm must assign the same sequence to A[t] and A[t′]. Thus (a) ==> (b).

Case 3. t and t′ are Q-nodes. As in Case 2 let the children of t (respectively t′) be $\{t_r\}_{r=1}^{k}$ (respectively $\{t'_r\}_{r=1}^{k}$). Note that as in the previous case there must be a one-to-one correspondence between the t_r and t'_r such that subtrees rooted at corresponding nodes are L-equivalent. However, since children of Q-nodes can only be reversed, the only two possibilities are

$$\text{LABEL}[t] = \text{LABEL}[t'] \tag{2a}$$

$$T_i(t_r) \equiv_L T_j(t'_r) \tag{2b}$$

$$a[t_r] = a[t'_r] \tag{2c}$$

or

$$\text{LABEL}[t] = \text{LABEL}^r[t'] \tag{3a}$$

$$T_i(t_r) \equiv_L T_j(t'_{k+1-r}) \tag{3b}$$

$$a[t_r] = a[t'_{k+1-r}], \tag{3c}$$

where in both cases part (c) follows from the inductive hypothesis. In either case, the set of values {A1,A2} calculated in the algorithm for t must be the same as that calculated for t′. Then since A[t] and A[t′] are each chosen

to be the lexicographically smaller of A1 and A2, they must be equal. Thus (a) ==> (b).

This completes the proof that (a) ==> (b).

Next we show that (b) ==> (c). Note from the algorithm that if A[t]=A[t´] we can decompose these strings to deduce that

$$\text{LABEL}[t]=\text{LABEL}[t']. \tag{4}$$

$$t \text{ and } t' \text{ have the same type, and} \tag{5}$$

$$a[t_r] = a[t'_r], \text{ for } 1 \leq r \leq k \tag{6}$$

where the sequence $t_1,t_2,\ldots,t_k$ (respectively $t'_1,t'_2,\ldots,t'_k$) is the sequence of children of t in T_i (respectively of t´ in T_j) after the pass through LOOP for nodes of level m. By the inductive hypothesis (6) yields

$$T'_i(t_r) =_L T'_j(t'_r). \tag{7}$$

Then (4), (5), and (7) yield

$$T'_i(t) =_L T'_j(t'),$$

completing the proof that (b) ==> (c).

Part (c) ==> (a) since all transformations made to trees during the algorithm are equivalence transformations. Thus so far we have shown that

$$(a) \iff (b) \iff (c).$$

Finally, (b) <==> (d) by an argument identical to that used in the basis. This completes the induction. []

Lemma 4. Algorithm 3 may be implemented to run in O(n) time.

Proof. Let $nnodes_m$ be the number of nodes at level m; let $nchar_m$ be the number of vertices whose characteristic node is at level m. It is easy to establish that for each level m, a bound on the total length of the sequences A[t] at level m, as well on their range of values, is

$$O(nnodes_{m-1} + nnodes_m + nchar_m). \qquad (8)$$

We will now show that the time used in the m^{th} pass tnrough LOOP is bounded by (8). First consider the three statements beginning with that labeled INDICES. Recall that to calculate the indices, we must sort the sequences A for nodes at level m. If we use Algorithm 3.2 in [1], the time for the sort is bounded by the total length of the sequences plus the maximum value of an element of a sequence. Thus (8) gives a time bound. When we actually perform the sort, the

sequences A[t] are used as keys in records which also contain the value of t, so that when the sequences are sorted, the t's are also ordered. It is then a straightforward task to number the sequences in lexicographically increasing order, assigning duplicates the same number, and thus to find the index to be assigned to a[t] for each t in time bounded by (8).

Now consider the time spent in LOOP for each type of node. (As in [1], we note that a list of vertices at each level may be obtained in linear time by a preorder traversal of the tree.) Statement Cloop clearly takes constant time. The implementation of block Ploop is rather indirect; we use the same trick used in the tree isomorphism algorithm in [1]. First set a list CHILDLIST for each P-node to the empty list. Recall that we have available, from the previous pass through LOOP, a list of all nodes t at level m-1 in order of nondecreasing A[t]. Scan this list in order; for each node t with a P-node parent p, add t to the end of CHILDLIST[p]. When this has been done, each CHILDLIST appears in nondecreasing order, so block Ploop can clearly be done in $O(nnodes_{m-1}+nnodes_m)$ time; thus the bound of (8) holds.

Finally, consider the cost of block Qloop. Recall the method discussed earlier for calculating $LABEL^r[q]$ from LABEL[q]. If q is a node with k children, then a bound on the number of pairs in LABEL[q] is $O(|CHAR^{-1}(q)|)$, and a bound on the range of values in LABEL[q] is O(k). Thus the calculation of $LABEL^r[q]$ can be performed in $O(|CHAR^{-1}(q)|+k)$ time through

the use of Algorithm 3.1 of [1]. Summing over all Q-nodes at level m gives (8). The remaining steps of block Qloop can easily be done, for all Q-nodes at level m, in time bounded by (8). Thus the time spent in the block labeled Qloop during this pass is bounded by (8).

We have now shown that each part of LOOP can be performed in time bounded by (8); summing over all m, we obtain O(N + n), which is linear. []

Theorem 4. Interval graph isomorphism may be decided in O(n+e) time.

Proof. Given interval graphs G and G', proceed as follows. First construct the trees T(G) and T(G'); as mentioned earlier, this can be done in O(n + e) time [4]. Use the algorithms presented in this section to obtain corresponding proper canonical labeled PQ-trees T and T'. Determine whether these are L-identical; this can be done in linear time by a preorder scan of trees. By Theorem 3, T and T' are L-identical if and only if G and G' are isomorphic.

[]

Note that we have done more than decide isomorphism efficiently; we have produced a compact (O(n) space) canonical representation for interval graphs. Note that if we have already computed the canonical representation, we may test two graphs for isomorphism in O(n) time; thus, for example, if we were searching a large list of graphs for

duplicates, we could precompute the canonical forms and then quickly make the necessary comparisons.

4. Chordal graph isomorphism is hard.

It is interesting to ask whether the efficient test for isomorphism outlined above may be extended to a larger class of graphs. One class of graphs which might be considered is the class of chordal graphs. As mentioned in the introduction, the class of chordal graphs properly contains the class of interval graphs [12]. We now show that it would be optimistic indeed to try to extend the isomorphism algorithm presented here to cover the class of chordal graphs. The method is very similar to that used in [9] to prove that isomorphism for certain other graphs is as hard as for arbitrary graphs.

Theorem 5. Arbitrary graph isomorphism is polynomially reducible to chordal graph isomorphism. (See [11] for more information about polynomial reducibility.)

Proof. Define a mapping M from an arbitrary graph $G=(V,E)$ to a graph $G'=(V',E')$ as follows. To avoid confusion, elements of V' will be called points and elements of E' will be called lines. V' contains a point for each vertex and each edge of G; call these v-points and e-points respectively. E' contains a line connecting each pair of v-points as well as a line connecting each v-point and e-point whose corresponding

elements in G are incident. Formally,

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

$$E' = \{\{v,w\} \mid v \text{ and } w \text{ are adjacent vertices in } G\} \cup$$
$$\{\{v,u\} \mid v \in V,\ u \in E, \text{ and } v \text{ is incident with } u\}.$$

See Figure 9. It is apparent that this mapping may be carried out in polynomial time.

Next we show that G' is chordal. Consider any cycle of length greater than three in G'. We examine two cases.

Case 1. The cycle contains only v-points. Then it certainly has a chord, since all v-points are adjacent.

Case 2. The cycle contains e-point. Then the two points adjacent to this point must be v-points, and hence are adjacent; thus we have the desired chord.

Now assume that G has at least four vertices; this creates no serious loss of generality, since we are only concerned with the asymptotic behavior of the algorithms. We will show that G' contains enough information to enable us to reconstruct G, up to isomorphism. First note that we can tell which points of G' are v-points and which are e-points: since all v-points are adjacent, all have degree at least equal to n-1, which is greater than 2; on the other hand, an e-point always has degree 2, since it always is adjacent to exactly two v-points. Finally, we can tell whether vertices of G are adjacent by checking whether the corresponding v-points are adjacent to a common e-point. Thus we can reconstruct G from

G' up to isomorphism.

We may therefore reduce the problem of testing isomorphism of G_1 and G_2 to the problem of testing isomorphism of $M(G_1)$ and $M(G_2)$, establishing the theorem. []

References

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

[2] K. S. Booth, PQ-tree algorithms, Ph. D. dissertation, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, California, 1975. (Also available as UCRL-51953 from Lawrence Livermore Laboratory, Livermore, California, 1975).

[3] K. S. Booth, Problems polynomially equivalent to graph isomorphism, Technical Report CS-77-04, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 1977.

[4] K. S. Booth and G. S. Lueker, Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms, JCSS 13, No. 3 (December 1976), pp. 335-379.

[5] M. Fischer and R. Ladner, private communication.

[6] D. R. Fulkerson and O. A. Gross, Incidence matrices and interval graphs, Pacific J. Math. 15 (1965), pp. 835-855.

[7] F. Gavril, The intersection graphs of subtrees in trees are exactly the chordal graphs, J. Comb. Theory, 16 (1974), pp. 47-56.

[8] G. Hajós, Über eine Art von Graphen, Internationale Math. Nachrichten, 11 (1957), p. 65.

[9] D. Hirschberg and M. Edelberg, On the complexity of computing graph isomorphism, Technical Report TR-130, Computer Science Laboratory, Department of Electrical Engineering, Princeton University, Princeton, N. J., August 1973.

[10] J. E. Hopcroft and R. E. Tarjan, Isomorphism of planar graphs, in Complexity of Computer Computations, R. E. Miller and J. W. Thatcher, eds., Plenum Press, New York, 1972, pp. 131-152.

[11] R. M. Karp, Reducibility among combinatorial problems, in Complexity of Computer Computations, R. E. Miller and J. W. Thatcher, eds., Plenum Press, New York, 1972, pp. 85-104.

[12] C. G. Lekkerkerker and J. Ch. Boland, Representation of a finite graph by a set of intervals on the real line,

Fund. Math. 51 (1962), pp. 45-64.

[13] A. Lempel, S. Even, and I. Cederbaum, An algorithm for planarity testing of graphs, in Theory of Graphs: International Symposium: Rome, July, 1966, P. Rosenstiehl, ed., Gordon and Breach, New York, 1967, pp. 215-232.

[14] G. S. Lueker, Efficient algorithms for chordal graphs and interval graphs, Ph. D. Dissertation, Program in Applied Mathematics and Department of Electrical Engineering, Princeton University, Princeton, N. J., 1975.

[15] F. S. Roberts, Discrete Mathematical Models, with Applications to Social, Biological and Environmental Problems, Prentice-Hall, Englewood Cliffs, N. J., 1976.

[16] D. J. Rose, R. E. Tarjan, and G. S. Lueker, Algorithmic aspects of vertex elimination on graphs, SIAM J. Comput. 5 (1976), pp. 266-283.

[17] S. M. Young, Implementations of PQ-tree algorithms, Masters Thesis, Department of Computer Science, University of Washington, Seattle, Washington, 1977.

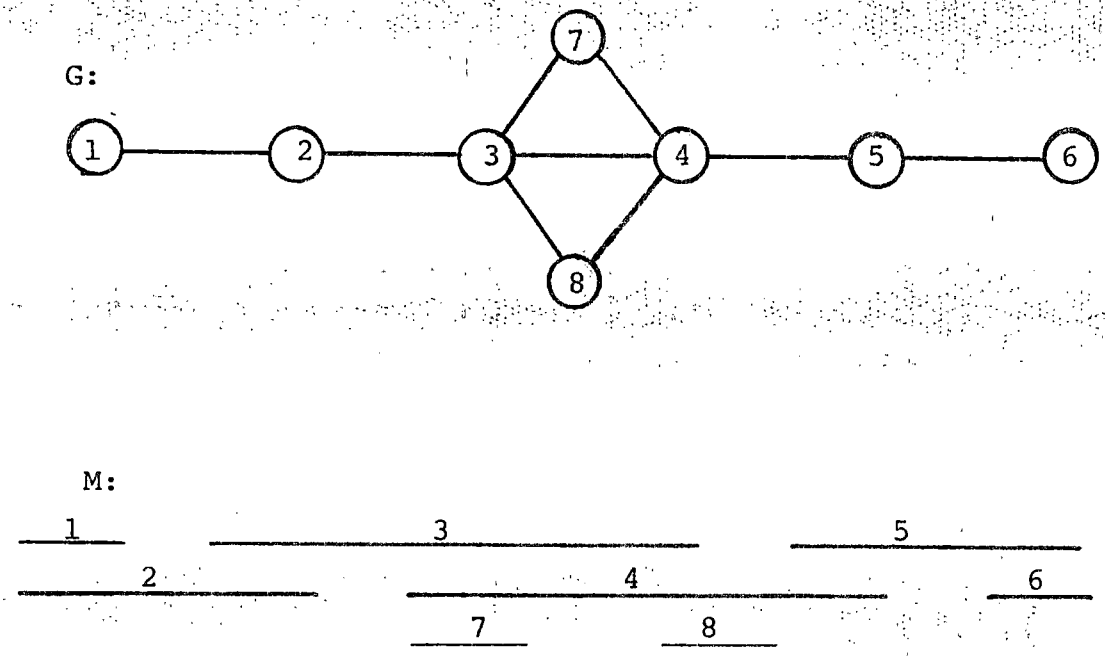


Figure 1. An interval graph G and an intersection model M for G.

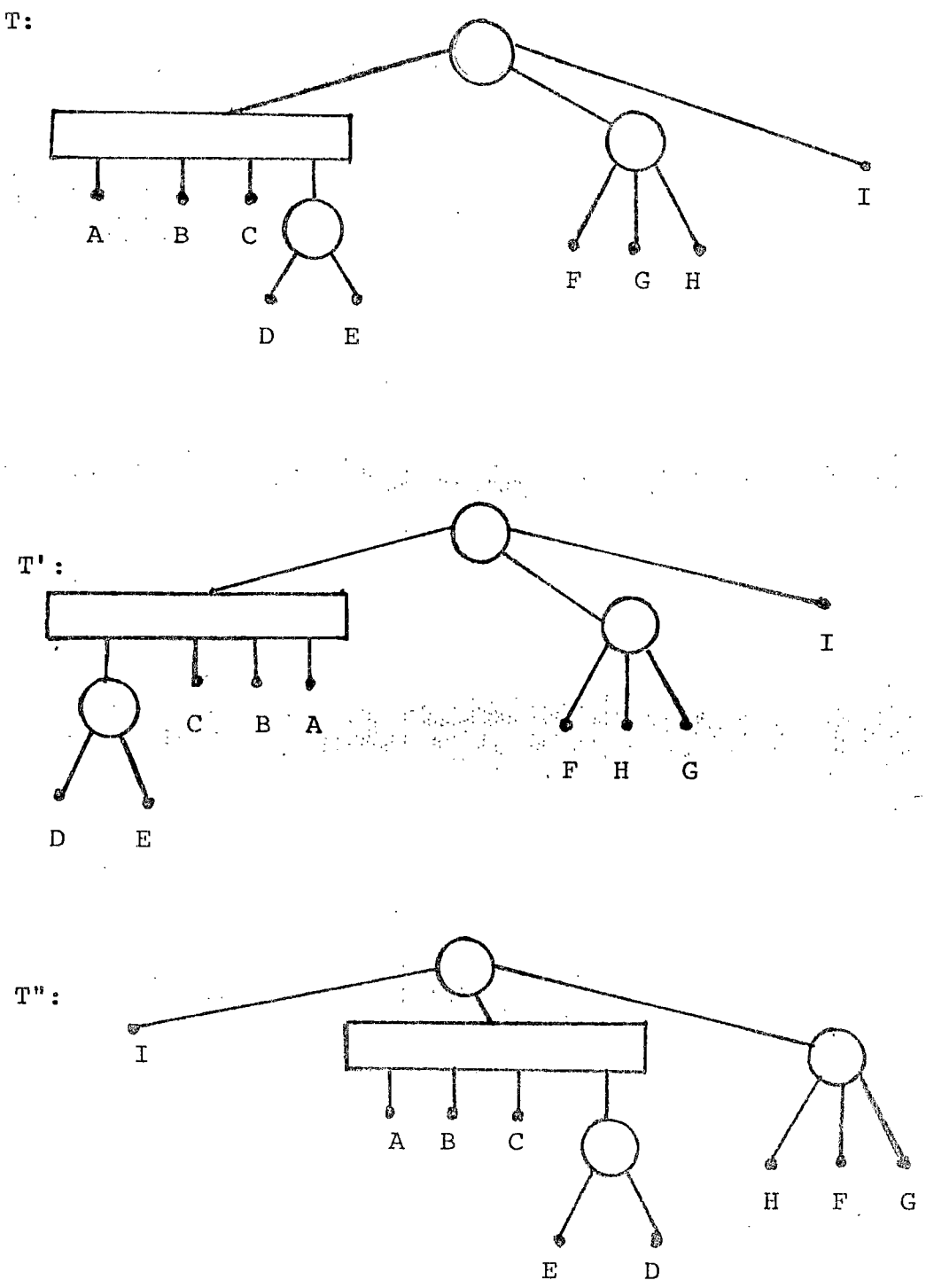


Figure 2. A PQ-tree T and two equivalent trees T' and T". We may conclude that ABCDEFGHI, DECBAFHGI, and IABCEDHFG are elements of CONSISTENT(T).

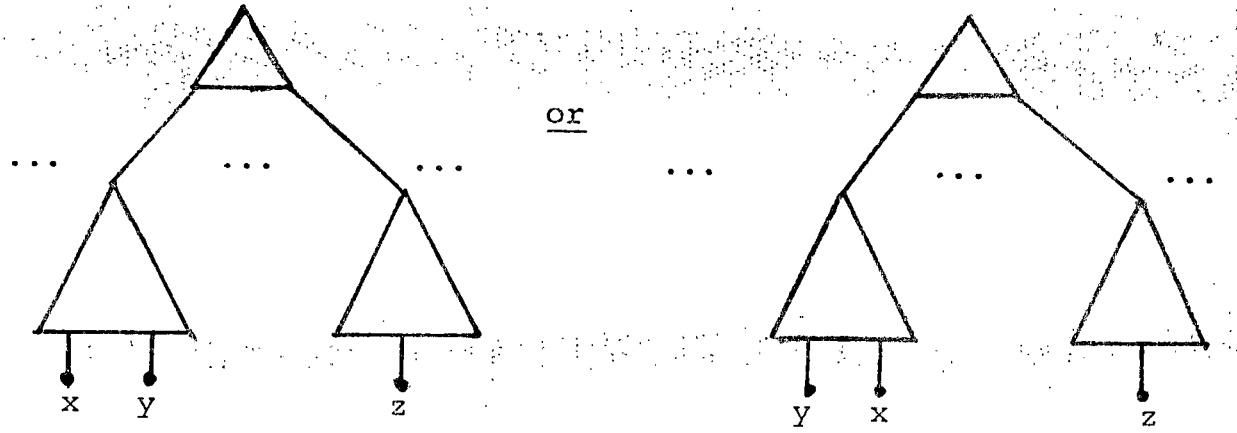


Figure 3. Two different ways to orient T_2 by reversing the frontier of a subtree.

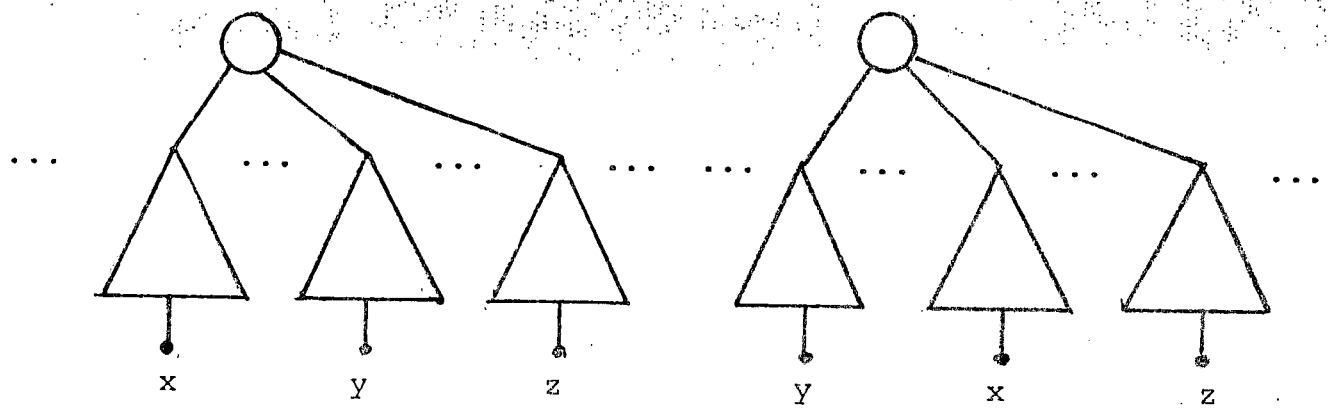


Figure 4. Two different ways to orient T_2 by permuting subtrees of a P-node root.

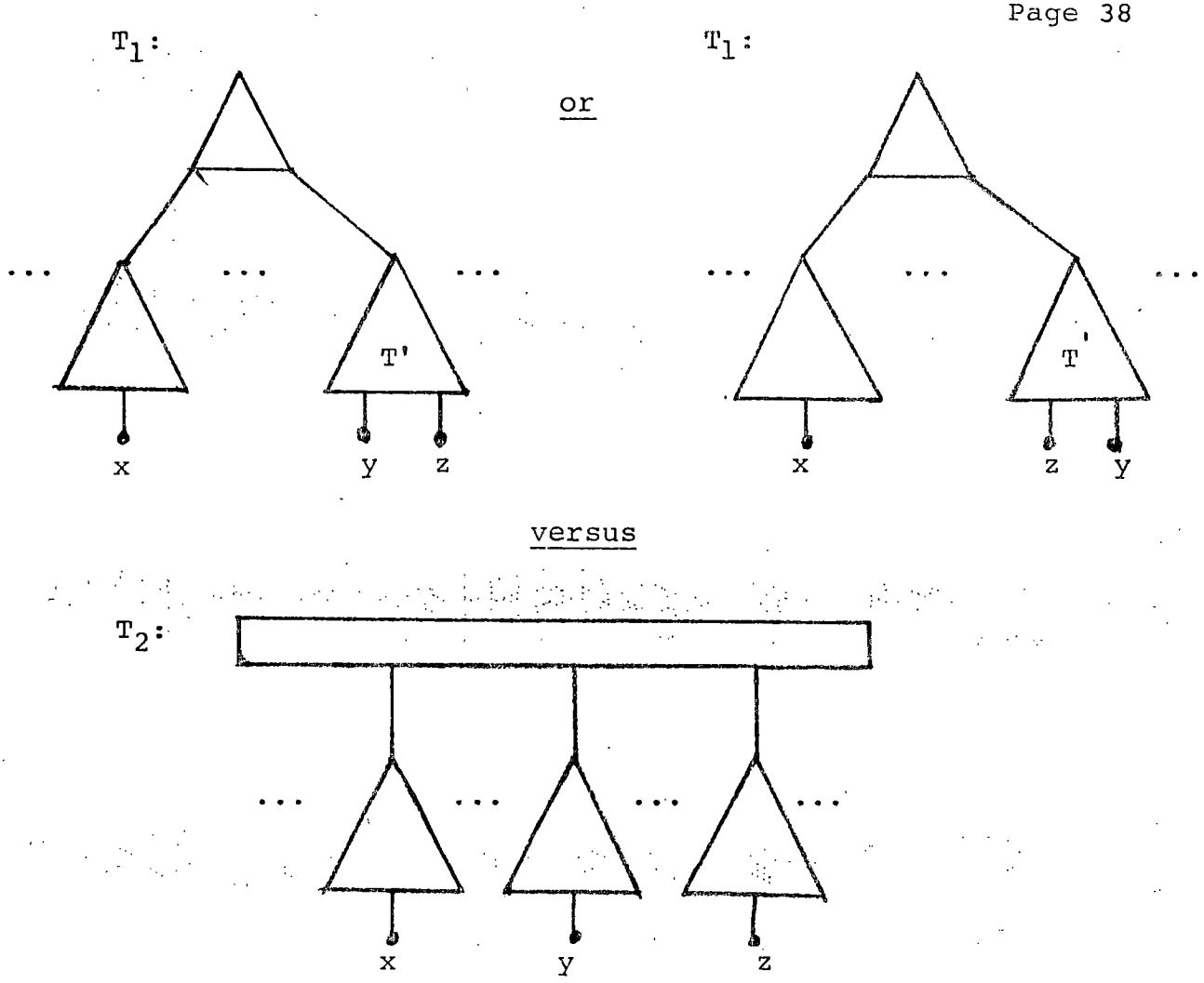


Figure 5. Two possible frontiers for T_1, at least one of which is inconsistent with T_2 having a Q-node root.

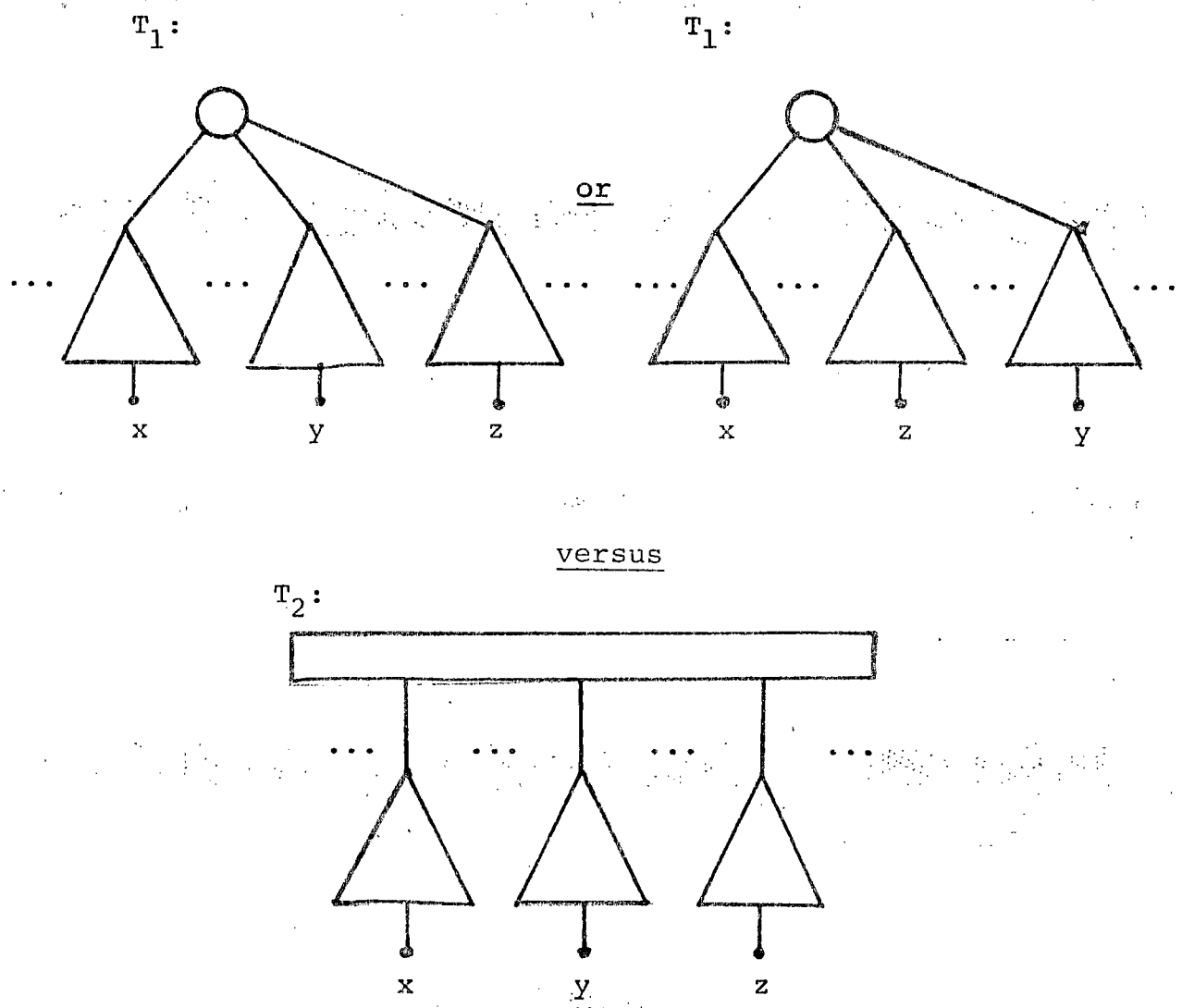
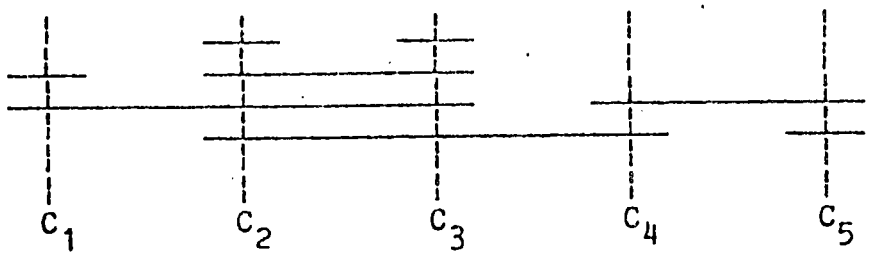
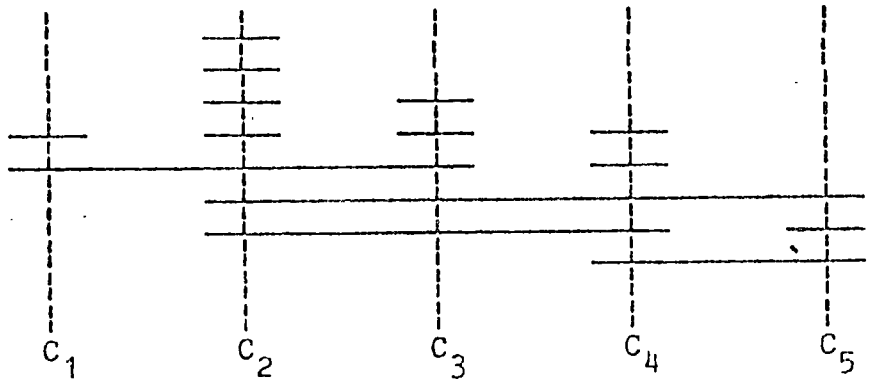


Figure 6. Two possible frontiers for T_1 (with a P-node root), at least one of which is inconsistent with T_2 having a Q-node root.

A model for G:



A model for G′:



A PQ-tree T for G or G':

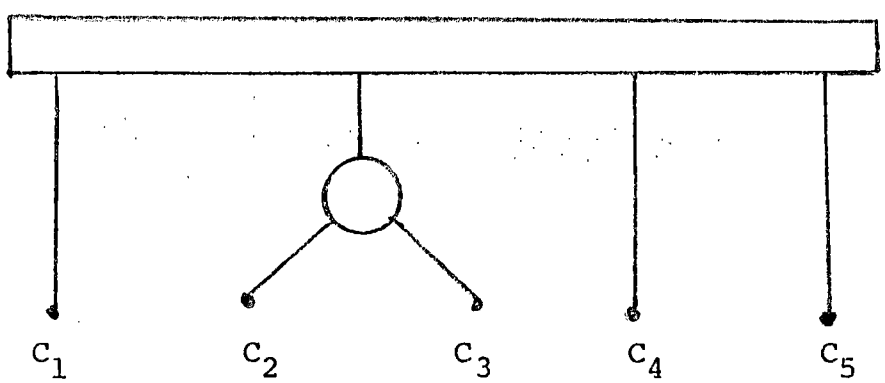


Figure 7. G and G' are not ismorphic, but T is a proper PQ-tree for either. For brevity and clarity, graphs are represented in this figure by their intersection models. Cliques are indicated by showing a point on the real line which is contained in all of the intervals corresponding to the vertices of the clique.

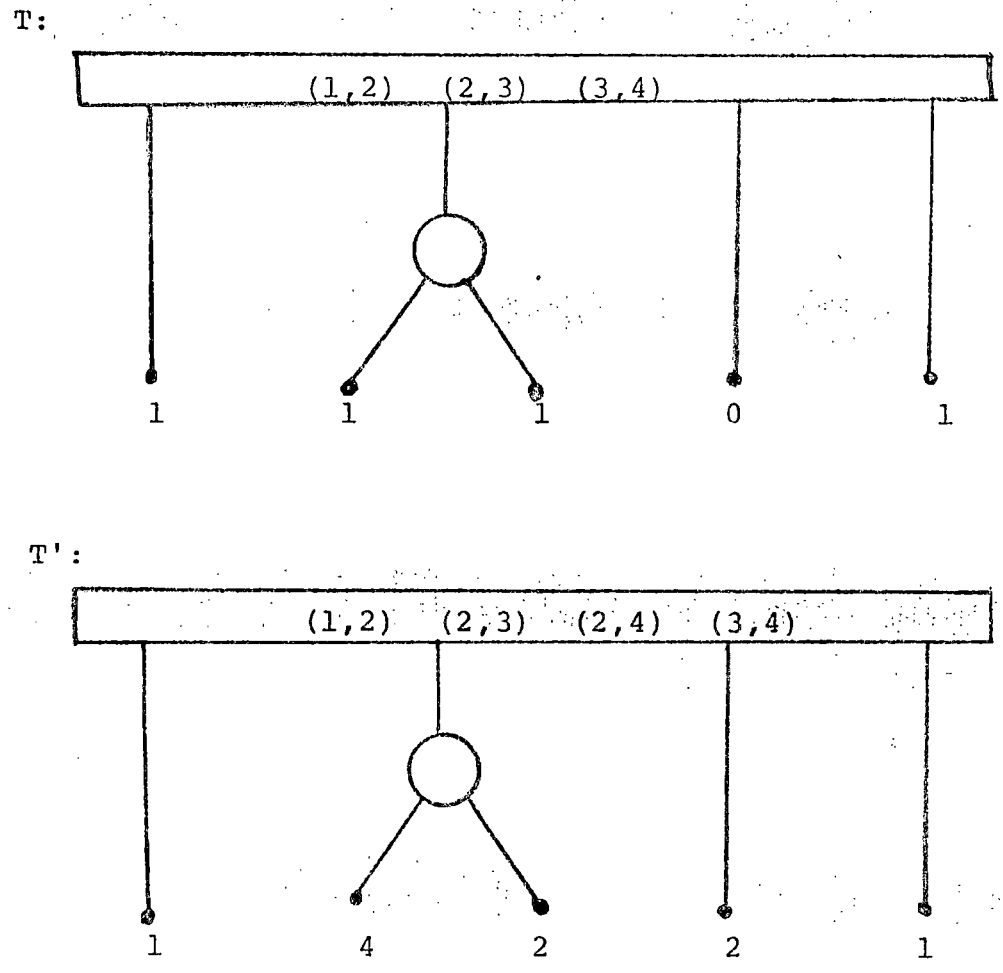


Figure 8. Labelled PQ-trees T and T' for G and G', respectively, of Figure 7.

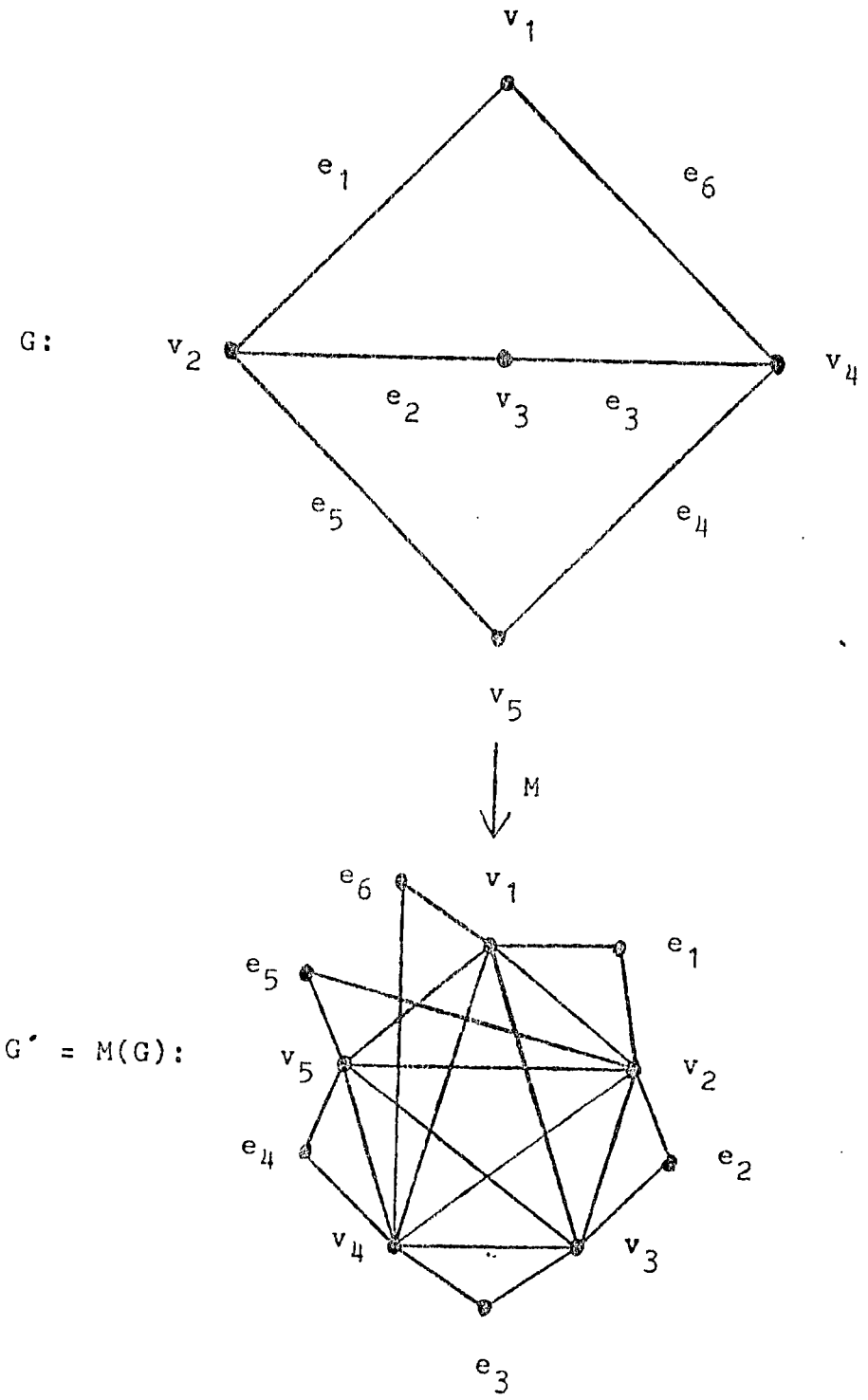


Figure 9. Example of the mapping M used in the reduction of arbitrary graph isomorphism to chordal graph isomorphism.