Testing for the Consecutive Ones Property, Interval Graphs, and Graph Planarity Using PQ-Tree Algorithms

KELLOGG J. BROWN

Computer Systems Division, Lawrence Livermore Laboratory, Livermore, California 94550

AND

GEORGE S. LUTKER

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

Received December 1975; revised July 1976

A data structure called a PQ-tree is introduced. PQ-trees can be used to represent the permutation of a set \( U \) in which various subsets of \( U \) occur consecutively. Efficient algorithms are presented for manipulating PQ-trees. Algorithms using PQ-trees are then given which test for the consecutive ones property in matrices and for graph planarity. The consecutive ones test is extended to a test for interval graphs using a recently discovered fast recognition algorithm for chordal graphs. All of these algorithms require a number of steps linear in the size of their input.

I. Introduction

A data structure called a PQ-tree is introduced here as an aid in solving three problems related to finding permissible permutations of a set \( U \). The permissible permutations are those in which certain subsets \( A \subset U \) occur as consecutive subsequences.
for \(m \times n\) matrices having \(f\) nonzero entries. Solutions to this problem have applications in a number of different fields. They also lead to efficient recognition and isomorphism tests for interval graphs.

The interval graph test discussed in Section 5 is a speeded-up version of an earlier algorithm of Fulkerson and Gross [8]. Their algorithm includes a test for graph chordality. Chordal graphs are discussed in [5] and [23]. A new algorithm based on lexicographic breadth-first search performs a chordality test in linear time [21, 24]. Combining a fast-reduction algorithm with the efficient chordality test results in an interval graph test whose overall time bound is \(O(n + e)\) steps for graphs having \(n\) vertices and \(e\) edges. Both chordal and interval graphs are related to Gaussian elimination, and can be used to analyze the efficiency of elimination schemes for sparse symmetric positive definite matrices [23, 24, 29].

The final application of PQ-trees is to the recognition of planar graphs. This is a classical graph theory problem for which linear algorithms have been presented in the literature [14]. The algorithm given in Section 6 is a revised version of an existing algorithm due to Lempel, Even, and Cederbaum [20]. The new version uses the reduction algorithm for PQ-trees. The result is an \(O(n)\) test for planarity, given a graph with \(n\) vertices.

The test for the consecutive ones property has been implemented in the programming language Pascal by Lader, Fischer, and Young [18]. Their implementation contains quite a bit of originality, since it was accomplished using only the brief description of the present work given at the 1975 SIGACT Conference.

2. PQ-Trees

The first order of business is to precisely define the data structure which is proposed and to relate the formal definitions to the informal remarks just given. After this preliminary the basic reduction algorithm for refining the class of permissible permutations is illustrated. The question of efficiency for this algorithm is put off until Section 3. A number of implementation details are also postponed until then.

Given a universal set \(U = \{a_1, a_2, \ldots, a_n\}\), the class of PQ-trees over that set is defined to be all rooted, ordered trees [1] whose leaves are elements of \(U\) and whose internal (nonleaf) nodes are distinguished as being either \(P\)-nodes or \(Q\)-nodes. As an illustration, each of the following three operations will construct a legal PQ-tree.

1. Every element \(a_i \in U\) is a PQ-tree whose root is the element. The tree consists of only a single leaf and is drawn as the element itself.

2. If \(T_1, T_2, \ldots, T_k\) are all PQ-trees then the structure shown in Fig. 1 is a PQ-tree whose root is a \(P\)-node.

3. If \(T_1, T_2, \ldots, T_k\) are all PQ-trees then the structure shown in Fig. 2 is a PQ-tree whose root is a \(Q\)-node.

A \(P\)-node is drawn as a circle, with its children (the trees \(T_1, T_2, \ldots, T_k\)) drawn below it.

A \(Q\)-node is drawn as a rectangle, with its children drawn below it.

These operations always produce leaves which are elements of \(U\) and internal nodes which are either \(P\)-nodes or \(Q\)-nodes, as required by the definition of a PQ-tree. The only difference between a \(P\)-node and a \(Q\)-node is the way in which the children are treated. This distinction between \(P\)-nodes and \(Q\)-nodes will become clearer shortly. One further set of restrictions is made on the nodes of a PQ-tree. A PQ-tree is proper exactly when each of the following three conditions holds.

1. Every element \(a_i \in U\) appears precisely once as a leaf. This is because PQ-trees are supposed to represent permutations of a set, so it does not make sense for an element to appear more than once or to not appear at all.

2. Every \(P\)-node has at least two children. This rules out long chains of nodes having only a single child. Scanning chains is very costly and should be avoided like the plague.

3. Every \(Q\)-node has at least three children. This again eliminates chains but also serves a more technical purpose. As explained below, there is no real distinction between a \(P\)-node and a \(Q\)-node if there are only two children. It is convenient to remove this redundancy.

These conventions serve to clean up some minor details by guaranteeing unique PQ tree representations for the classes of permutations being studied [3]. For this reason, only proper PQ trees will be considered throughout the remainder of this paper. At some points during the algorithms described later, the actual trees constructed are not in fact proper. This does not violate the convention just established because whatever impurity is introduced is swiftly removed. After a complete reduction the final tree is always proper.
more inhibited. The restriction to simple reversal means that the same two children will always be endmost and all of the others will be interior. In addition, each interior child of a $Q$-node always has the same two immediate siblings. These facts are useful when manipulating PQ-trees so the properties endmost and interior will be used later to classify nodes for processing.

There is only one operation on PQ-trees. Given a subset $S \subseteq U$, and a tree $T$, a new tree is needed whose consistent permutations are exactly the original permutations in which the leaves selected by $S$ occur in some order as a consecutive sequence. There is a natural way to obtain such a tree. The new tree is called the $S$-reduction of $T$. It is denoted by REDUCE($T$, $S$). The $S$-reduction can be constructed from the original tree by examining the tree node-by-node. The reduction procedure given below defines the reduced tree. The fact that REDUCE($T$, $S$) is actually the PQ-tree which represents the desired class of permutations is proven later in Theorem 1.

The procedure REDUCE is a more complete version of the inner loop for the reduction algorithm given in Section 1. It is still not the entire implementation. Many details have yet to be discussed. The procedure applies a sequence of templates to the nodes of a PQ-tree. Each template has a pattern and a replacement. If a node matches the template's pattern, the pattern is replaced within the tree by the template's replacement. The value of the procedure is a new PQ-tree. It is the null tree if the original tree could not be reduced for the set specified.

**PQ-tree procedure REDUCE($T$, $S$):**

begin
  initialize QUEUE to empty;
  for each leaf $x \in U$ do place $x$ onto QUEUE;
  while $|\text{QUEUE}| > 0$ do
    begin
      remove $X$ from the front of QUEUE;
      if some template applies to $X$ then
        substitute the replacement for the pattern in $T$
      else
        begin
          $T =$ null tree;
          exit from do
        end;
    end;
    if $S \subseteq \{Y \mid X$ is an ancestor of $Y \}$ then exit from do;
    if every sibling of $X$ has been matched then
      place the parent of $X$ onto QUEUE
    end;
  return $T$
end

Each template specifies a local change within the tree. Only the node being matched and its children are altered. The patterns to which nodes are matched depend upon the set $S$ and the frontier of the subtree rooted at the particular node for which a match is being sought. The matched pattern is selected by examining the node and its children after the children themselves have been matched. This is the reason that the parent of a node is only queued if the node is actually the last sibling to be matched. Queuing enforces the child-before-parent discipline. The bottom-up strategy is important because it allows information to be propagated from the leaves to the internal nodes in a controlled manner.

A node $X$ is said to be full if all of its descendants are in $S$; $X$ is said to be empty if none of its descendants are in $S$. If some but not all of the descendants are in $S$, $X$ is said to be partial. Nodes are said to be pertinent if they are either full or partial. The pertinent subtree of $T$ with respect to $S$, denoted PERTINENT($T$, $S$), is the subtree of maximum height whose frontier contains all of $S$. The pertinent subtree and its root are unique. The root of the pertinent subtree is denoted by ROOT($T$, $S$).

It is usually not the root of the entire tree $T$. Figure 6 shows the pertinent subtree for the tree appearing in Fig. 3. The set used in this example is $S = \{e, f, j, k\}$. ROOT($T$, $S$) is the $O$-node at the top.

**Fig. 6.** The pertinent subtree for the PQ-tree in Fig. 3 when $S = \{e, f, j, k\}$. 

Pattern-matching is very simple for leaves. There are only two templates. Either a leaf is a member of $S$ or else the leaf is not a member of $S$. In either event there is no change to the tree when the replacement is made except to note that the leaf in question is labeled either full or empty.

Matters are more complicated for internal nodes. The goal is to ensure that after replacement the frontier of the tree rooted at the matched node has all of its pertinent leaves occurring as a consecutive subsequence of the frontier. There are a number of different patterns which arise. Some of these cases cannot accommodate the goal. They result in a failure return from the matching procedure. If no template is found, a failure return is made.

There are a few cases for $P$-node too. Any $P$-node which has all of its children labeled empty can be labeled empty and no change is necessary. Similarly if all of a node's children are labeled full then the node can be labeled full and left alone.

The templates for these cases are shown in Fig. 4. Nodes which are labeled as empty are drawn normally (those on the left side in this example) and nodes which are empty.
must be on either one side or the other. The replacement for Template P3 guarantees this.

Confessions are in order. A number of details have been glossed over and should be carefully examined. The first point is that, as warned previously, the replacement for Template P3 creates an improper PQ-tree because the Q-node which is labeled partial has only two children. This is all right, however. Since there is at least one other pertinent leaf which is not a descendant of this Q-node, there will eventually be three children. This will happen later, as ancestors of the current node are matched.

The second point is that Template P3 has some alternate forms. These occur when there is only one full child or one empty child. Replacements for these special cases are shown in Fig. 10.

![Fig. 10. Alternate replacements for Template P3.](image)

All of the loose ends have been tied up now. But of course there are still more cases to come. With the introduction of partial nodes there is the added possibility that one of the children is partial. If a P-node has exactly one partial child, then the P-node is labeled singly partial. There are two cases, depending upon whether or not the P-node is ROOT(T, S). Figure 11 shows the template for the root and Fig. 12 shows the template for other P-nodes. These are analogous to Template P2 (Fig. 8) and Template P3 (Fig. 9).

![Pattern.]  
![Replacement.](image)

![Pattern.](image)  
![Replacement.](image)

If there are less than two empty or full children, the corresponding P-node is eliminated from the replacement for Template P4 or Template P5. The corresponding replacements are similar to those shown in Fig. 10. It may even happen that there are no empty children or no full children. In these cases both the missing children and their P-node are deleted from the replacement.

The final legal template for P-nodes is the pattern having precisely two singly partial children. The P-node being matched is labeled doubly partial to denote the fact that two partial children exist. In this case the P-node must be the root of the pertinent subtree. If it is not the root there is no way that the matching can continue. This is easy to see. Each partial child has an empty end and a full end. The only possibility is the replacement shown in Fig. 13. If there is any other pertinent leaf it is impossible to make it consecutive with the leaves in the subtree rooted at the doubly partial node.

![Pattern.](image)  
![Replacement.](image)

![Pattern.](image)  
![Replacement.](image)

![Pattern.](image)  
![Replacement.](image)
Obviously, $\text{CONSISTENT}(U, S)$ is exactly the class of permutations in which the elements of $S$ occur in a consecutive subsequence. Two special cases are worth noting. $T(U, U)$ is the universal PQ-tree because the two $P$-nodes violate the rules for proper trees so they are collapsed to one $P$-node. $T(V, (v))$ is another way of specifying the null tree. If there are no leaves there cannot be any internal nodes. This notation will be used later as a convenient abbreviation.

**Theorem 1.** Let $T$ be any PQ-tree and let $S \subseteq U$ be any subset of the universal set.

\[
\text{CONSISTENT}(\text{REDUCE}(T, S)) = \text{CONSISTENT}(T) \cap \text{CONSISTENT}(T(U, S)).
\]

**Proof.** To show that the two classes of permutations are actually the same it is sufficient to prove that each is contained within the other. Equality then follows. The first step is to demonstrate that the left-hand side is contained within the right-hand side. This has two parts.

Let $\pi$ be any permutation in the class $\text{CONSISTENT}(\text{REDUCE}(T, S))$. If such a permutation exists the tree $T$ must have been successfully reduced, otherwise $\text{REDUCE}(T, S)$ would be the null tree. By definition the reduced form of $T$ has an equivalent tree whose frontier is $\pi$. Let this equivalent tree be $T'$. The basic idea underlying the proof is to construct a tree $T''$ which is equivalent to $T$ and which has $\pi$ as its frontier. This is accomplished by undoing the template-matching which reduced $T$, but without undoing the equivalence transformations which were used during the template-matching. The process of applying a template has two stages; the pattern must be matched, possibly requiring an equivalence transformation, and then the replacement made. Consider what happens when the templates are applied in the reverse order from which they are applied during the reduction algorithm. If the replacement is undone but the equivalence transformation used to match the pattern is retained, the entire template-matching process can be run in reverse. The result is a tree $T''$ which is equivalent to the original tree $T$ but which has the same frontier as $T'$. This common frontier is $\pi$, hence $\pi \in \text{CONSISTENT}(T)$.

Next we show that $\pi$ is also in $\text{CONSISTENT}(U, S)$, that is, that the elements of $S$ appear consecutively in $\pi$. By inspection of the templates which can apply to the root, we see that after the root is matched there will be a node $A$ in the tree such that

\[a_1, \ldots, a_k, a_{k+1}, \ldots, a_n \in \pi.\]

**PQ-Tree Algorithms**

(a) the descendants of $A$ include all of $S$, and
(b) $A$ is either a full $P$-node or a $Q$-node all of whose pertinent children are full and appear consecutively.

From this it follows that all elements of $S$ must appear consecutively in the frontier of any tree equivalent to the reduced tree.

Finally we show that $\text{CONSISTENT}(T) \cap \text{CONSISTENT}(T(U, S))$ is contained in $\text{CONSISTENT}(\text{REDUCE}(T, S))$. That is, if a permutation $\pi$ is in the consistent set of the original tree and happens to have the elements of $S$ appearing consecutively, then $\pi$ is also in the consistent set of the reduced tree. Let $T'$ be a tree equivalent to $T$ and have $\pi$ as its frontier. The fact that $S$ appears consecutively in $\pi$ implies that no node in the pertinent subtree, except the root, can have more than one partial child; the root may have at most two partial children. Note also that after any partial node other than the root is matched, it becomes a $Q$-node; moreover, its sequence of children, examined from left to right (or possibly from right to left), will consist of a sequence of full nodes followed by a sequence of empty nodes. These observations guarantee that each node matches one of the templates. Next note that for each template, the fact that $S$ is consecutive in the frontier of $T$ implies that the replacement can take place with no change to the frontier of the tree. Thus $\pi$ is in $\text{CONSISTENT}(\text{REDUCE}(T, S))$. Then since $T'$ and $T$ are equivalent, and since template matching preserves equivalence, $\pi$ is also in $\text{CONSISTENT}(\text{REDUCE}(T, S))$.

This first theorem explains why PQ-trees are interesting. They can be used to represent all of the permutations in which each set in a family of subsets occurs as a consecutive subsequence. If the initial PQ-tree is the universal tree for some set $U$ and $S$-reduction restricts the consistent permutations to be exactly those in which $S$ is consecutive. Multiple reductions on a family of sets produce the class of permutations in which every one of the sets is consecutive. This is precisely the operation needed for the applications which follow.

**3. Efficient Implementation.**

The template-matching algorithm of Section 2 is purposely lacking in details. No claim is made that it represents an efficient technique for performing reductions on PQ-trees. This matter will be addressed here. There are two problems to be faced when implementing the reduction algorithm. The program must decide which templates to apply and then it must apply them. These two actions are not independent. They interact with each other and this interaction must be taken into account.

The only real constraint imposed by the template-matching approach is that all of the children of a node must be matched before the parent is matched. This require-
Both have valid pointers to the parent which can be given to the node being processed. No change in the blockcount is required. This situation is diagrammed in Fig. 20.

![Fig. 20. Unblocked interior node leaving the blockcount unchanged.]

If one of the node’s siblings is already blocked there is no increase in the blockcount. Two nodes are now blocked. Figure 21 shows this case.

![Fig. 21. Blocked interior node leaving the blockcount unchanged.]

A similar situation is shown in Fig. 22. This time one of the siblings is already unblocked so the new node is immediately given a parent pointer. Here too the blockcount is not changed.

More interesting case is when both siblings are blocked. Again the new node is blocked, but there is an important difference. The blockcount must be decremented by one because two blocks of blocked nodes have merged into a single block. This is shown in Fig. 23.

![Fig. 22. Unblocked interior node leaving the blockcount unchanged.]

![Fig. 23. Blocked interior node which decrements the blockcount.]

The final case is the most interesting. If one of the siblings is blocked and one of the siblings is unblocked the node being processed is unblocked since it can obtain a valid parent pointer. But more can be done. The pointer can be passed through the entire block of blocked siblings. Thus the blockcount decrements and a number of orphaned nodes suddenly receive parents. This is shown in Fig. 24. All of the cases shown in Figs. 19-24 apply only to interior children of Q-nodes. When an endmost child is found it already has a parent pointer and thus it is automatically unblocked. A check is made to see if the immediate sibling is blocked. If it is, a scan similar to that employed in Fig. 24 is used to pass the parent pointer across the block of blocked siblings. The blockcount is decremented by one if there are any blocked siblings which become unblocked.

![Fig. 24. Unblocked interior node which decrements the blockcount.]

Using this strategy, every pertinent node will have a valid parent pointer at the beginning of the second pass. The interior pertinent children can be counted at the same time they receive parent pointers so that the parent will have a correct count during the second pass. Care must be taken to initialize all of the nodes to unmarked and to reinitialize them after each reduction, but this can be easily handled at no additional cost.

If the tree cannot be reduced it may happen that some interior nodes remain blocked. This can be detected during the bubbling up because the blockcount will be greater than zero. This is not quite correct, though. If the root of the pertinent subtree is a Q-node and all of its pertinent children are interior then a blockcount
PERTINENT_CHILD_COUNT. A count of the number of pertinent children currently possessed by a node. This count is initially zero and is incremented by one each time a child of the node is processed during the bubbling up. During the matching pass the count is decremented by one each time a child is matched. The node is spurious for matching when the pertinent child count reaches zero during the second pass.

PERTINENT_LEAF_COUNT. A count of the number of pertinent leaves which are descendants of this node. This field is built up during the second pass as each child of the node is matched. It is the sum of the pertinent leaf counts for all of the pertinent children.

TYPE. A designation telling whether the node is a leaf, a P-node, or a Q-node.

A complete version of the bubbling up pass is now given. It can be translated into most higher-level languages in a straightforward manner using common programming techniques [1].

PQ-tree procedure BUBBLE(T, S);
begin
initialize QUEUE to be empty;
BLOCK_COUNT := 0;
BLOCKED_NODES := 0;
OFF THE TOP := 0;
for X ∈ S do place X onto QUEUE;
while |QUEUE| + BLOCK_COUNT + OFF THE TOP > 1 do
begin
if |QUEUE| = 0 then
begin
T := T(0, 0);
exit from do
end;
remove X from the front of QUEUE;
MARK(X) := "blocked";
BS := |S| - IMMEDIATE_SIBLINGS(X) | MARK(Y) := "blocked";
US := |S| - IMMEDIATE_SIBLINGS(X) | MARK(Y) := "unblocked";
if |US| > 0 then
begin
choose any Y ∈ US;
PARENT(X) := PARENT(Y);
MARK(X) := "unblocked"
end
else if IMMEDIATE_SIBLINGS(Y) < 2 then MARK(Y) := "unblocked";
if MARK(Y) = "unblocked" then
begin
Y := PARENT(Y);
if |BS| > 0 then
begin
LIST := the maximal consecutive set of blocked siblings adjacent to Y;
for Z ∈ LIST do
begin
MARK(Z) := "unblocked";
PARENT(Z) := Y;
PARENT_CHILD_COUNT(Y) := PARENT_CHILD_COUNT(Y) + 1
end
end
end;
if Y = nil then OFF THE TOP := 1
else
begin
PERTINENT_CHILD_COUNT(Y) :=
PERTINENT_CHILD_COUNT(Y) + 1;
if MARK(Y) = "unmarked" then
begin
place Y onto QUEUE;
MARK(Y) := "queued"
end
end;
BLOCK_COUNT := BLOCK_COUNT + |BS|;
BLOCKED_NODES := BLOCKED_NODES + LIST
end
end
end
end
if not TEMPLATE $P(X)$ then
  if not TEMPLATE $P(Y)$ then
    if not TEMPLATE $P(Z)$ then
      if not TEMPLATE $Q(X)$ then
        if not TEMPLATE $Q(Y)$ then
          if not TEMPLATE $Q(Z)$ then
            $T := T(n, c)$; exit from do
          end
        end
      end
    end
  end
end

return $T$

All that remains is to describe the templates. There are a few things to be careful of here also. Note that when Template $P_5$ is applied there are a number of empty children which receive a new parent. In order to achieve the linear bounds to follow, we must avoid having to manipulate fields within these empty children. To do this, we use their original parent as the “new” $P$-node which groups the empty children. This avoids changing any parent pointers of empty nodes.

Procedures for Templates $P_5$ and $Q_2$ are shown here in detail; the other template procedures are similar. Note the order in which templates are tried. This information can be used to simplify some of the procedures because information is implicitly known about previous templates which failed.

**Boolean procedure TEMPLATE $P_5(X)$:**

begin
  if TYPE($X$) = $P$-node then return false;
  if |PARTIAL_CHILDREN($X$)| = 1 then return false;
  $Y :=$ the unique element in PARTIAL_CHILDREN($X$);
  $F_X :=$ the unique element in ENDMOST_CHILDREN($Y$) labeled “empty”;
  $F_Y :=$ the unique element in ENDMOST_CHILDREN($Y$) labeled “full”;
  comment the following statement may be performed in time on the order of
  the number of pertinent children of $X$ through the use of
  the CIRCULAR_LINK fields;
  if $Y$ has an empty sibling then $F_X :=$ an empty sibling of $Y$;
  comment $Y$ will be the root of the replacement;
  PARENT($Y$) := PARENT($X$);
  PERTINENT_LEAF_COUNT($Y$) := PERTINENT_LEAF_COUNT($Y$);
  LABEL($Y$) := “partial”;
  PARTIAL_CHILDREN(PARENT($Y$)) := PARTIAL_CHILDREN(PARENT($Y$))\{$Y$};
  remove $Y$ from the list of children of $X$ formed by the CIRCULAR_LINK fields;

  if |IMMEDIATE_SIBLINGS($X$)| = 0 then
    replace $X$ by $Y$ in the list of children of PARENT($X$) formed by the
    CIRCULAR_LINK fields
  else
    begin
      replace $X$ by $Y$ in the list of children of PARENT($X$) formed by the
      IMMEDIATE_SIBLINGS fields;
      if $Y$ ENDMOST_CHILDREN(PARENT($Y$)) then
        ENDMOST_CHILDREN(PARENT($Y$)) := ENDMOST_CHILDREN(PARENT($Y$))\{$Y$};
    end;
  end;

  if |FULL_CHILDREN($X$)| > 0 then
    begin
      if |FULL_CHILDREN($X$)| = 1 then
        let $ZF$ be the unique element in FULL_CHILDREN($X$) and remove
        $ZF$ from the CIRCULAR_LINK list of which it is currently a member
      else
        begin
          create a new $P$-node called $ZF$; LABEL($ZF$) := “full”;
          for each node $W$ in FULL_CHILDREN($X$) do
            begin
              remove $W$ from the CIRCULAR_LINK list of
              which it is currently a member;
              PARENT($W$) := $ZF$;
            end;
          end;
        end;

        set the CIRCULAR_LINK fields of the nodes in
        FULL_CHILDREN($X$) to form a doubly-linked circular list;
        CHILD_COUNT($ZF$) := |FULL_CHILDREN($X$)|
    end;

  PARENT($ZF$) := $Y$;
  IMMEDIATE_SIBLINGS($ZF$) := IMMEDIATE_SIBLINGS($Y$)\{$ZF$};
  IMMEDIATE_SIBLINGS($Y$) := ($Y$); ENDMOST_CHILDREN($Y$) := ENDMOST_CHILDREN($Y$)\{$ZF$};
end;

NUMBER_EMPTY := CHILD_COUNT($Y$);
if NUMBER_EMPTY > 0 then
  begin
  if NUMBER_EMPTY = 1 then
    $ZE := EN$ else

Proof. As indicated above, the algorithm can be implemented so that the work performed in matching any node \( X \) is on the order of one plus the number of pertinent children of \( X \). Summing this over all nodes matched, we obtain \( O(\text{PRUNED}(T, S)) \),

These two bounds can be combined into a single bound which will suffice to prove linearity for the applications in the following sections. The proved reduction algorithm is simply BUBBLE followed by REDUCE. The overall algorithm is as follows:

\[
\text{PQ-tree procedure REDUCTION}(U, S);
T := T(U, U);
\]

for each \( S \in \mathcal{S} \) do
begin
\[
T := \text{BUBBLE}(T, S);
T := \text{REDUCE}(T, S);
\]
end;
return \( T \)
end

One problem which has not been addressed explicitly here is the reinitialization of certain fields. For example, the MARK fields should be set to "unmarked" at the beginning of each pass. One convenient way to do this is to maintain a list of all nodes which are modified during the reduction; after the reduction is complete one may scan this list and reset the fields of the nodes as required.

The next lemma is useful in obtaining linear bounds on the time used by PQ-tree algorithms. Define a unary node to be a node in the pertinent subtree which has just one pertinent child. Let \( \text{UNARY}(T, S) \) be the set of all unary nodes in \( T \) with respect to the set \( S \).

**Lemma 4.** The proved reduction algorithm requires only \( O(|S| + |\text{UNARY}(T, S)|) \) steps to reduce \( T \) with respect to \( S \).

**Proof.** There are only \( |X| \) leaves. Binary branching implies that there are at most \( O(|S|) \) unary nodes in \( \text{PRUNED}(T, S) \). The remainder of the proof follows directly from Lemmas 2 and 3.

The interesting form of this lemma is its application to the case of multiple reductions. The terms due to unary nodes within the trees can be eliminated by averaging their cost over all of the sets. Given a family of sets \( \mathcal{S} \) define \( \text{SIZE}(S) \) to be the sum of the sizes of all the sets in the family.

**Theorem 5.** The class of permutations in which each set of a family \( \mathcal{S} \) occurs as a consecutive subsequence can be computed in \( O(m + n + \text{SIZE}(S)) \) steps if \( T \) has \( m \) objects and \( S \) has \( n \) sets.

**Proof.** We use the algorithm REDUCTION presented above. One readily sees that the work outside of the calls to BUBBLE and REDUCE uses only \( O(m + n) \) time. From Lemma 4 the total work in all of the BUBBLE and REDUCE calls can be computed as the sum of two terms. One is the total contribution due to the sizes of the sets, this is just \( \text{SIZE}(S) \). The second term is the sum of \( |\text{UNARY}(T, S)| \) over all sets \( S \) in \( \mathcal{S} \). This term is a bit harder to bound. We begin by noting that a unary node cannot be the root of \( \text{PERTINENT}(T, S) \) and that its children cannot be empty or all full. Thus the only templates which can apply to unary nodes are \( P3 \), \( P5 \), and \( Q2 \). First consider Template 3. It is not hard to see that if \( P3 \) is applied more than twice, there must be three partial nodes in \( T \), none of which is an ancestor of any other, from which it follows that \( S \) cannot be made consecutive so REDUCE fails. We conclude that the total number of applications of \( P3 \) is \( O(m + n) \).

The bound on the number of applications of \( P5 \) and \( Q2 \) is obtained by a more indirect argument. The Template \( Q2 \) must be split into two subcases: let \( Q_2^1 \) be \( Q2 \) when no partial children are present and let \( Q_2^2 \) be \( Q2 \) when one partial child is present. By an argument like that used for \( P3 \), we see that \( Q_2^2 \) can be applied a total of \( O(m + n) \) times. Now let \( \text{NORM}(T) \) be the number of \( Q \)-nodes in \( T \) plus the number of nodes in \( T \) whose parent is a \( P \)-node. It is easy, though tedious, to verify the following facts.

(a) Initially, \( \text{NORM}(T) \) is at most \( m \).
(b) No template replacement increases \( \text{NORM}(T) \) by more than one.
(c) Templates \( P5 \) and \( Q2^1 \) reduce \( \text{NORM}(T) \) by at least one.

Now we have already seen that the number of applications of all templates except \( P5 \) and \( Q2^1 \) is \( O(m + n + \text{SIZE}(S)) \). Then since \( \text{NORM}(T) \) is clearly nonnegative, (a), (b), and (c) imply that the number of applications of \( P5 \) and \( Q2^1 \) is \( O(m + n + \text{SIZE}(S)) \). This completes the proof.

The bound shown in Theorem 5 will be used in Sections 4 and 6 to prove linear bounds for algorithms which use PQ-tree reduction.

4. CONSECUTIVE ONES

A 0, 1-matrix \( M \) has the consecutive ones property for columns if its rows can be permuted so that in each column all of the ones are consecutive [8]. This means that a permutation of the rows is desired for which no two ones within a single column are separated by a zero in that same column. A number of related properties can also be defined such as the circular ones property in which the ones are allowed to "wrap around" from the bottom of a column to the top [25].
ones property iff \( M' \) has the consecutive ones property. \( M' \) can be computed from \( M \) in \( O(m + n + f) \) steps.

Proof. Choose a row having a minimum number of ones. Construct the matrix \( M' \) as in the previous lemma. Each row will have no more than the number of ones it originally contained plus the number of ones produced by complementation.

This at most doubles the total number of ones in the matrix.

\( M' \) can be constructed by scanning a list of the nonzero entries of \( M \). The complementation is easily performed using list handling techniques. Since each one is only processed a fixed number of times, independent of the size of \( M \), the total time required is \( O(m + n + f) \).

Corollary 9. An \( m \times n \) \((0, 1)\)-matrix specified by its \( f \) nonzero entries can be tested for circular ones in \( O(m + n + f) \) steps.

Proof. By Lemma 8 the computation of \( M' \) is within the desired time bound. The number of edges in \( M' \) is \( f' \leq 2f \) which is \( O(f) \). The rest of the work is just the consecutive ones test so the total work is \( O(m + n + f) \).

There are a number of generalizations for the consecutive ones property, including some \( NP \)-complete problems associated with finding matrices which approximate these properties. Further related results are surveyed in [3].

5. INTERVAL GRAPHS

A graph \( G = (V, E) \) is an interval graph iff there is a 1-1 correspondence between its vertices and a set of intervals on the real line such that two vertices are adjacent iff the corresponding intervals have a nonempty intersection. The set of intervals is called an intersection model for \( G \). Hapman was the first to mention these graphs in the literature [13]. Since then interval graphs have been related to problems in biology [2], psychology [22], and traffic light sequencing [27]. They are also related to Gaussian elimination schemes for sparse symmetric positive definite matrices [29].

Characterizations for interval graphs have been given in [8, 11, 19]. Polynomial recognition algorithms exist based on each of the three characterizations. All have worst-case time bounds of at least \( O(n^3) \) for graphs with \( n \) vertices. Using the reduction algorithm for \( P \bar{Q} \) trees this can be lowered to \( O(n + \epsilon) \) for graphs with \( n \) vertices and \( \epsilon \) edges. The characterization in [8] is the first occurrence in the literature of the consecutive ones property. Consecutive ones play a key role in interval graph recognition, as evidenced by the following theorem which also appears in [8].

Within a directed graph \( G = (V, E) \) a clique is a completely connected subgraph. The dominant cliques are those which are maximal with respect to set inclusion.

**Theorem 11 (Fulkerson, Gross).** \( G = (V, E) \) is an interval graph iff its dominant-clique vs vertex matrix has the consecutive ones property for columns.

This characterization suggests an algorithm for recognizing interval graphs. The first step is to compute the dominant cliques. The set of nonzero entries can be compacted for the matrix. This task is made considerably easier by the fact that every interval graph is a chordal graph [5]. These are just those graphs for which every cycle \( C \) of length greater than three there exists an edge connecting two vertices of \( C \) which are not consecutive in \( C \). The algorithms given in [8, 11, 19] each test for chordality as a precondition to being an interval graph. They do not have obvious linear-time implementations.

Gavril has shown how to test chordality in \( O(n^{4/3}) \) steps [9]. His algorithm is based on the fact that a chordality test can be turned into a matrix multiplication, for an appropriately defined matrix. The time bound then follows from Strassen's result [28].

A more efficient algorithm, based on a lexicographic breadth first search of the graph, has been developed in [21, 24]. Lexicographic breadth-first search is very similar to the idea used in [4] and [26] for solving certain scheduling problems.

The dominant-clique vs vertex matrix can be constructed in \( O(n + \epsilon) \) steps. Using this technique, a useful property of chordal graphs is the fact that the dominance-chlique vs vertex matrix never has more than \( O(n + \epsilon) \) nonzero entries [8]. Putting all of these pieces together the following algorithm emerges. Let \( G = (V, E) \) be an undirected graph.

**Boolean procedure INTERVAL(V, E);**

**begin**

**if** \( G \) is not chordal **then** return false;

let \( U \) be the dominant cliques of \( G \);

\( T := T(U, U) \);

**for** \( e < 1 \) **do**

**begin**

let \( S \) be the set of cliques containing \( e \);

\( T := \text{BUBBLE}(T, S) \);

\( T := \text{REDUCE}(T, S) \);

if \( T = T(e, e) \) **then** return false

**end**;

**return true**

**end**
begin
  \( S := \) the set of edges whose higher-numbered vertex is \( j \); 
  \( T := \) BUBBLE(\( T, S \)); 
  if \( T = T(S, S) \) then return false; 
  \( S := \) the set of edges whose lower-numbered vertex is \( j \); 
  if \( \text{ROOT}(T, S) \) is a \( Q \)-node 
    then replace the full children of \( \text{ROOT}(T, S) \) and their 
        descendants by \( T(S, S) \); 
  else replace \( \text{ROOT}(T, S) \) and its descendants by \( T(S, S) \); 
  \( U := U \cup S \cup S \); 
end; 
return true 
end

The numbering defined in Lemma 13 ensures that neither \( S \) nor \( S' \) is ever empty. As usual we must always make sure that the tree is proper. Therefore when part of \( T \) is replaced by \( T(S, S) \) a little care must be taken. For example, if \( \text{ROOT}(T, S) \) is a \( Q \)-node and after the replacement it has only two children, it is changed to a \( P \)-node. Also, if after the replacement any node has only one child, it is removed from the tree.

The planarity algorithm explained here is equivalent, except for some minor details, to the algorithm presented in [20]. A proof of correctness is given in [20]. It is left to the reader to establish that the version presented here implements the same algorithm. The proof is not hard, but a full discussion of the original algorithm is beyond the scope of this paper. Instead the algorithm will be illustrated by running it on the simple graph shown in Fig. 26.

The tree is initialized outside of the main loop to be a universal tree with a leaf for each edge directed out of vertex 1. This first tree is shown in Fig. 27. Any permutation of the edges is legal. This corresponds to the fact that there are no constraints on the graph. Only vertex 1 has been positioned. As more vertices are added the freedom to rearrange vertices is diminished. This is reflected in the \( PQ \)-tree by the ordering imposed upon the edges.

![Fig. 27. Initial PQ-tree for planarity testing.](image1)

The inner loop is repeated three times. At each iteration the edges entering a particular vertex are forced to be consecutive. For the first iteration, \( j = 2 \), the reduction is trivial. There is only one such edge. It is \((1, 2)\). The tree is already \((1, 2)\)-reduced. After the edges entering vertex 2 are known to be consecutive within the tree they are replaced by a single \( P \)-node having a child for each edge directed out of vertex 2. This substitution produces the next tree shown in Fig. 28. Notice that the set \( U \) has changed. The edge \((1, 2)\) has been removed and the edges \((2, 3), (2, 4), \) and \((2, 5)\) have been added.

![Fig. 28. PQ-tree after first iteration of inner loop.](image2)

This first iteration is complete. The second iteration processes vertex 3, since \( j = 3 \). The current tree must be \((1, 3), (2, 3)\)-reduced so that all of the edges coming into vertex 3 are consecutive. The result is the tree in Fig. 29. This time the reduction is nontrivial.

![Fig. 29. PQ-tree after reducing edges entering vertex 3.](image3)

There are two edges which enter vertex 3. They are replaced with a \( P \)-node. One of the special cases occurs, because the root of the pertinent subtree is a \( Q \)-node. This causes no problem, though, and the substitution yields the tree of Fig. 30.

![Fig. 30. PQ-tree after substituting edges leaving vertex 3.](image4)
378 BOOTH AND LUTER

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


6. S. Even, personal communication.


14. G. Jungnickel, Representation of a collection of finite sets as intervals on a line, manuscript, Courant Institute, New York University, 1974.


17. B. R. LaRue, M. J. Fischer, and S. Young, private communication.

18. C. J. Leckerness and J. C. Boland, Representation of a finite graph by a set of intervals on the real line, *Fund. Math.* 51 (1962), 45-64.


