JOURNAL OF COMPUTER AND SYSTEM SCIENCES 13, 335–379 (1976)

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

KELLOGG S. BOOTH[†]

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

AND

GEORGE S. LUEKER[‡]

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 permutations 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.

1. 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 $S \subset U$ occur as consecutive sub-

* This report was prepared as an account of work sponsored by the United States Government. Neither the United States nor the United States Energy Research and Development Administration, nor any of their employees, nor any of their contractors, subcontractors, or their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately-owned rights.

[†] Work performed under the auspices of the U.S. Energy Research and Development Administration under Contract No. W-7405-Eng-48.

[‡] Research supported by a National Science Foundation Graduate Fellowship and NSF Grant GJ-1052 while in the Program in Applied Mathematics and the Department of Electrical Engineering at Princeton University.

sequences. A PQ-tree represents a class of permissible permutations. As the elements of each new subset S are constrained to appear together the number of permissible permutations is reduced. The corresponding PQ-tree operation is called reduction with respect to the set S. An efficient algorithm for computing the reduction of a PQ-tree is presented. Its time complexity is shown to be linear in the size of the input. The reduction algorithm is used to build linear algorithms for testing the consecutive ones property in matrices, for recognizing interval graphs, and for testing a graph for planarity.

Before defining the data structure it is worth explaining just what it is that these three problems have in common. The answer is that in each case the situation arises in which there is a set U of objects (the rows of a matrix, the dominant cliques of a graph, or the edges of a graph) and it is required, for reasons specific to the problem, that only certain permutations of these objects be permitted. Exactly which permutations are legal is determined by restrictions that are also dependent upon the specific problem, but for the problems considered here the restrictions always take the form "given the legal permutations obtained so far, permit only those permutations in which the objects of the set S (some subset of U) occur consecutively." That is to say, disallow a permutation in which two objects belonging to S are separated by an object not belonging to S.

As an example of such restrictions, suppose that an information retrieval system is being built. The following model considered by Ghosh [10] is a useful abstraction. A data base consists of a set U of records on a disk. The retrieval system answers certain queries concerning information stored in these records. A query can be thought of as a subset $S \subset U$, where S consists precisely of those records which are needed to answer the query. Suppose that the records are arranged on the disk so that each query S has its set of records stored in consecutive locations. Any query can be answered with only a single positioning move and transmission time directly proportional to the amount of data actually required for the query. Within this model consecutive storage arrangements are optimal and algorithms for finding the arrangements are of practical interest.

The retrieval example can be illustrated with the following simple data base having three records. Let the set of records be $U = \{A, B, C\}$. If the only queries are $\{A\}$, $\{B\}$, and $\{C\}$ then *any* of the six possible permutations of the records is acceptable. Suppose that the system must also handle the query $\{A, B\}$. Now, because records A and B must always be consecutive, there are only four permissible permutations: ABC, BAC, CAB, and CBA. Adding another query $\{B, C\}$ reduces the number of possible storage arrangements even further. Only ABC and CBA are permissible. So far, so good. But if the system must also accomodate the query $\{A, C\}$, all hope is lost! There is *no* permutation of the three records which simultaneously satisfies all of the constraints; either A and B are not consecutive, B and C are not consecutive, or else A and C are not consecutive.

This particular retrieval system is easy to analyze. As the number of records becomes large and the number of queries becomes even larger, there is at least the possibility that determining whether a legal permutation exists might prove to be a hopelessly complex combinatorial task. The answer, of course, is that this is not the case. A polynomial-time algorithm has already appeared in the literature [8]. The algorithm presented here achieves a linear running time. The improvement is due to the data structure. *PQ*-trees are devised to handle precisely the problem stated above, and to handle it efficiently.

An informal statement of the general problem is as follows. An algorithm must be devised which accepts as input a set of objects U and a family of subsets of those objects. The family is denoted by $\mathbb{S}$. The algorithm must decide whether there exists any permutation π such that the objects of each subset $S \in \mathbb{S}$ occur within π as a consecutive subsequence. Ideally the algorithm should also determine all permutations which satisfy these criteria. A general solution, written in Pidgin Algol similar to the style suggested in [1], is called reduction because it reduces the possible class of permutations to only those which satisfy the constraints.

```
**Boolean procedure** REDUCTION($U$, $\mathbb{S}$);
**begin**
  $\Pi := \{\pi \mid \pi$ is a permutation on $U\}$;
  **for** each $S \in \mathbb{S}$ **do**
    $\Pi := \Pi \cap \{\pi \mid$ all objects of $S$ are consecutive within $\pi\}$;
  **return** $\Pi \neq \varnothing$
**end**
```

The algorithm is easy to implement if each set is specified by a list of its objects. The only hard part is the inner loop. Given the current set of legal permutations those which do not satisfy the new constraint must be thrown away. This in fact is where all of the real work occurs. The next two sections develop machinery used to perform the inner loop efficiently.

The key to the algorithm's success is the introduction of an appropriate data structure for representing the entire class Π using only a modest amount of storage while still maintaining enough information to process the inner loop. One data structure which works is a *PQ*-tree. A similar idea appears in [20] as part of a planarity algorithm, but the representation uses formulas instead of trees and does not achieve a linear running time. *PQ*-trees are defined in Section 2 and an efficient algorithm for manipulating them is given in Section 3.

The first problem to which *PQ*-trees are applied is the consecutive ones property for matrices [8], which is discussed in Section 4. A permutation of the rows of a matrix is desired which places all of the ones within each column into consecutive form. This is a straightforward application of the basic reduction algorithm. Using the linear time bound for reduction, an $O(m + n + f)$ complexity bound is proven 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 Ladner, 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* $U = \{a_1, a_2, \ldots, a_m\}$, 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.

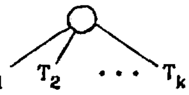

FIG. 1. A *P*-node.

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

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

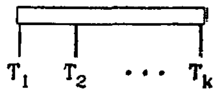

FIG. 2. A *Q*-node.

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 impropriety is introduced is swiftly removed. After a complete reduction the final tree is always proper.

Reading the leaves of a tree T from left to right yields its *frontier*, denoted by FRONTIER(T). The example which follows in Fig. 3 is a (proper) PQ-tree for the set $U = \{A, B, C, D, E, F, G, H, I, J, K\}$. The tree's frontier is the permutation $ABCDEFGHIJK$.

The frontier of a tree is obviously a permutation of the set U. The problem at hand involves an entire class of permutations Π. In order for PQ-trees to help they must be able to represent all of the permissible permutations.

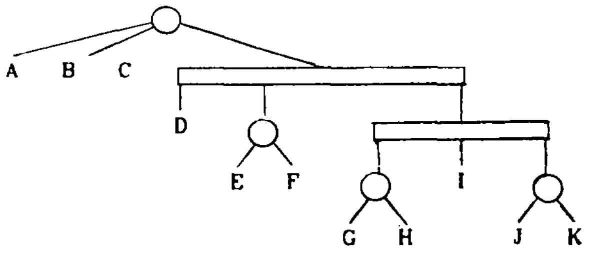

FIG. 3. A PQ-tree.

The definition of a PQ-tree does not suggest much about the order in which the children of a node should appear. Because the basic problem involves grouping all of the elements belonging to a particular set S it seems natural that internal nodes group their descendants. It also seems natural, in the absence of any other information, to leave the order within the group unspecified. Such freedom to reorder children means that a particular tree can frequently be redrawn by changing the left-to-right order of some of the children of some of the nodes. This freedom to rearrange can be formalized in the following manner.

Two trees are ***equivalent*** iff one can be transformed into the other by applying zero or more ***equivalence transformations***. Each transformation specifies a legal reordering of the nodes within a tree. There are only two types of equivalence transformations:

1. arbitrarily ***permute*** the children of a P-node,
2. ***reverse*** the children of a Q-node.

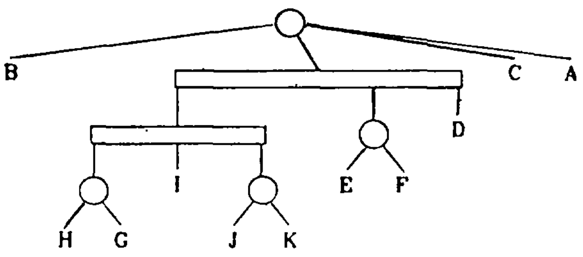

FIG. 4. A PQ-tree equivalent to the tree of Fig. 3.

The equivalence of two trees is written $T \equiv T'$. For the example in Fig. 3 there are 768 different trees in the *equivalence class*. One more of these trees is shown in Fig. 4. The new tree's frontier is *BHGIJKEFDCA*.

Every tree in an equivalence class has a different frontier. Conveniently an equivalence class of *PQ*-trees has exactly the frontiers which correspond to a class Π of permutations defined by rules of the form "keep all the elements consecutive for each set $S \in \mathbb{S}$."

A tree built only of leaves and *P*-nodes cannot represent the class $\Pi = \{ABC, CBA\}$ which occurs in the information retrieval example of Section 1. There are only three possible trees to check. None of these have the desired rearrangements. The necessity for both *P*-nodes and *Q*-nodes is thus obvious. The sufficiency of only two types, which may not be so obvious, is dealt with at the end of this section.

It is convenient to have a name for all of the frontiers which can be obtained by equivalence transformations on a tree. The set of *consistent permutations* for a tree is denoted by

$$\text{CONSISTENT}(T) = \{\text{FRONTIER}(T') \mid T' \equiv T\}.$$

One special tree is singled out and given a name. It is the *universal tree* which has a single *P*-node for its root and a leaf for every element of U. The universal tree has every possible permutation in its consistent set. A second definition is also useful. The *null tree*, which has no nodes at all and hence is not even a *PQ*-tree according to the rules, is admitted to the club anyway. By convention the null tree has no frontier and its set of consistent permutations is empty. Both trees are shown in Fig. 5 below. All other *PQ*-trees fall somewhere between these two extremes. It is shown in [3] that the classes of consistent permutations form a lattice whose smallest element corresponds to the null tree and whose largest element corresponds to the universal tree.

a_1
a_2
$\cdot\cdot$
a_m
Universal tree
Null tree

FIG. 5. The universal tree and the null tree.

This completes the cast of characters. They are the set $U = \{a_1, a_2, ..., a_m\}$ from which the leaves are chosen, two types of internal nodes, some rules for properly pasting things together, and two special trees which are given names—the universal tree and the null tree. There are a number of consequences for these definitions which are worth exploring.

The complete freedom with which the children of a *P*-node are permuted means that there is no implied left-to-right order among them. The children of a *Q*-node are 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 $S \subset U$ 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 fairly 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 $|$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;
      if $S \subset \{Y \mid X$ 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. Queueing 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 minimum 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 $S = \{E, I, J, K\}$. ROOT(T, S) is the Q-node at the top.

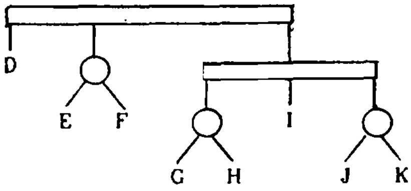

FIG. 6. The pertinent subtree for the PQ-tree in Fig. 3 when $S = \{E, I, J, K\}$.

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 cases which arise. Some of these cases cannot accomodate the goal. They result in a failure return from the matching procedure when no template is found.

There are a few easy cases for P-nodes 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. 7. Nodes which are labeled as empty are drawn normally (those on the left side in this example) and nodes which are labeled as full are drawn with shading (those on the right side). Only the children's labels matter. The children's types (leaf, P-node, or Q-node) do not matter during template-matching. All children are drawn as triangles regardless of their type.

Templates consist of two pieces. The first (top) piece is the pattern. The node being matched and its children must be equivalent to the pattern. If so the template applies and the second (bottom) piece specifies the replacement to make within the tree. For both of the templates in Fig. 7 the replacement is exactly the same as the pattern. Other templates are not so simple.

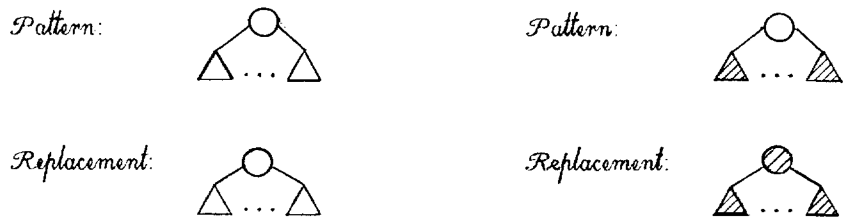

FIG. 7. Easy cases for P-nodes—Template $P0$ (left) and Template $P1$ (right).

The nontrivial cases occur when the children of a P-node have different labels. The situation immediately becomes more complicated. The full children have to be grouped together in order to ensure that all of the pertinent leaves (elements of S) are consecutive. The template in Fig. 8 accomplishes this task when the P-node is the root of the pertinent subtree. It is sufficient to simply gather all of the full children under a new P-node which then is made a child of ROOT(T, S). The root is left unlabeled because the template-matching is finished. The algorithm does not queue the parent of a matched node if all of the pertinent leaves have been accounted for. This is by definition true for ROOT(T, S).

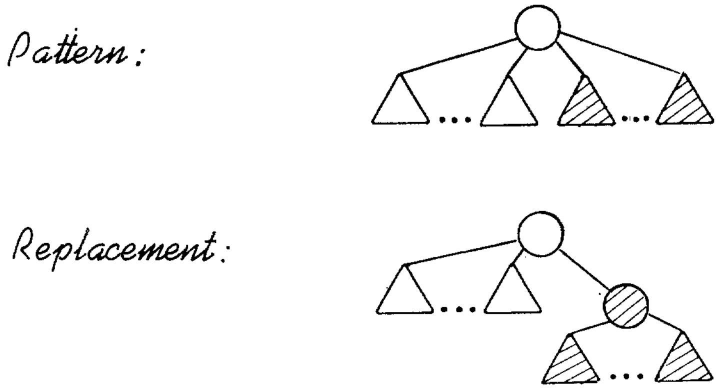

FIG. 8. Template $P2$ for ROOT(T, S) when it is a P-node.

A match for Template *P*2 exists if some equivalence transformation can be applied to ROOT(*T*, *S*) which will order the children of ROOT(*T*, *S*) in the same way as the children in the pattern. The transformation is applied to ROOT(*T*, *S*) and then the replacement is substituted into *T*. A single template can thus be used to match many different patterns.

Before the template of Fig. 8 is applied all of the children of the *P*-node are free to rearrange in any order whatsoever. After the replacement this is not the case. Empty children and full children may no longer intermingle. The reason for this should be obvious. If the final tree is to have all of the pertinent leaves consecutive along the frontier, no node having nonpertinent descendants can come between two nodes having pertinent descendants. Respecting this inevitability, the arrangement shown retains the most flexibility. Empty children are completely free to rearrange among themselves and the full children are likewise free. Moreover, the full children may appear anywhere within the empty children. This cannot hurt, since all of the pertinent leaves have been accounted for.

Most *P*-nodes are not the root of the pertinent subtree. After all, there is only one root. The normal case requires a little more care. If a *P*-node has some children which are labeled empty and some children which are labeled full then the *P*-node must be designated *singly partial*. This is a new label, different from either empty or full, which can only appear on an internal node. The template for this case is shown in Fig. 9. Partial nodes are partially shaded, with the shading indicating which children are full.

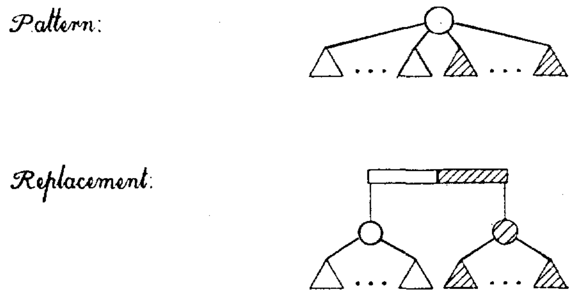

FIG. 9. Template *P*3 for a singly partial *P*-node which is not ROOT(*T*, *S*).

As before, a match exists if there is some equivalence transformation which causes the node being matched and its children to look like the pattern. Unlike Template *P*2, the replacement for Template *P*3 does not allow the full children of the *P*-node to appear just anywhere within the empty children. Since the *P*-node being matched is known not to be ROOT(*T*, *S*) there must be at least one other pertinent leaf which is not a descendant of the *P*-node. This leaf will eventually have to become consecutive with those leaves which are descendants. This implies that all of the full children must be on either one side or the other. The replacement for Template *P*3 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 *P*3 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 *P*3 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.

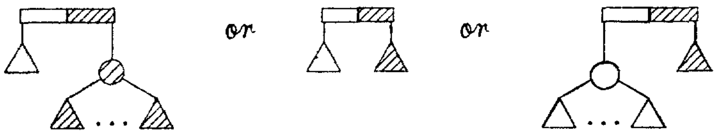

FIG. 10. Alternate replacements for Template *P*3.

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 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 *P*2 (Fig. 8) and Template *P*3 (Fig. 9).

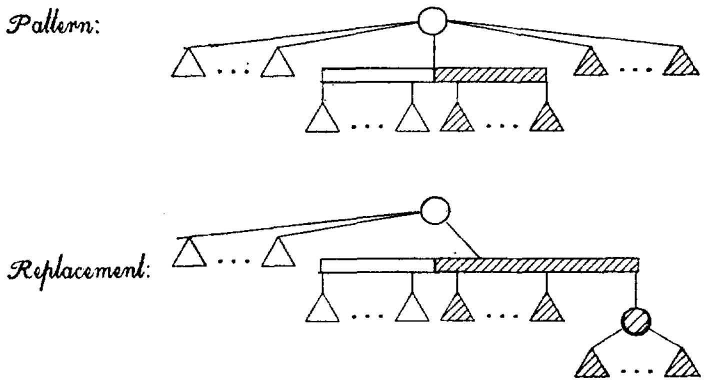

FIG. 11. Template *P*4 for ROOT(*T*, *S*) when it is a *P*-node with one partial child.

If there are less than two empty or full children, the corresponding *P*-node is eliminated from the replacement for Template *P*4 or Template *P*5. 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.

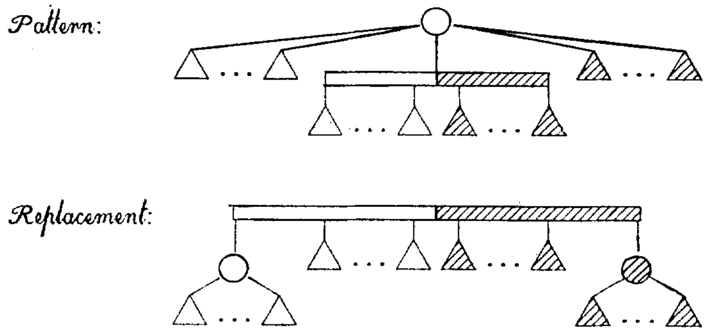

FIG. 12. Template *P*5 for a singly partial *P*-node, other than ROOT(*T*, *S*), with one partial child.

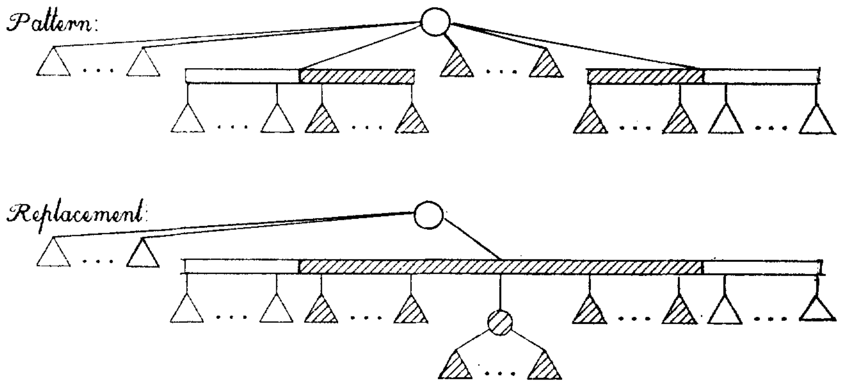

FIG. 13. Template *P*6 for ROOT(*T*, *S*) when it is a doubly partial *P*-node.

All other P-nodes not matching one of the templates just described are considered illegal. The template-matching process halts unsuccessfully and the tree is irreducible. These illegal cases correspond to P-nodes with more than two partial children. It is easy to see that no matter in what order three partial nodes occur, pertinent leaves which are descendants of the left and right nodes will be separated by nonpertinent leaves which are descendants of the middle node. This violates one of the constraints imposed by the sets, hence it is ruled illegal.

Q-nodes also have a number of different templates. The cases when all of the children are labeled identically are taken care of by Template $Q0$ and Template $Q1$, which are analogous to Template $P0$ and Template $P1$. There is no change to the Q-node except for the new label it receives. As with P-nodes the interesting situation is when some of the children have different labels. There are only two major cases which remain, but each has a number of subcases. The most general situations are illustrated.

A Q-node is singly partial if it has at most one singly partial child and if the left-to-right order of the children is that shown in Fig. 14. The rule which says that a template applies if any equivalence transformation of the node and its children matches the pattern allows only that the children of either Q-node may be reversed.

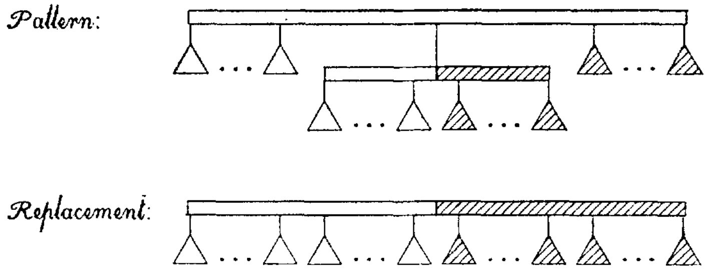

FIG. 14. Template $Q2$ for a singly partial Q-node.

Some or all of the empty, full, or partial children may be missing from the pattern. The only important requirement is that not all of the children are labeled identically (so Templates $Q0$ and $Q1$ do not apply). As before, children missing from the pattern are deleted from the replacement.

The final legal Q-node template is the doubly partial case. Just as for P-nodes, a doubly partial Q-node must be the root of the pertinent subtree. Up to two children may be singly partial (although there might not be any partial children at all) but the left-to-right order must be exactly that shown in Fig. 15, except for a possible reversal. Note that both endmost children must either be empty or partial; otherwise Template $Q2$ would apply.

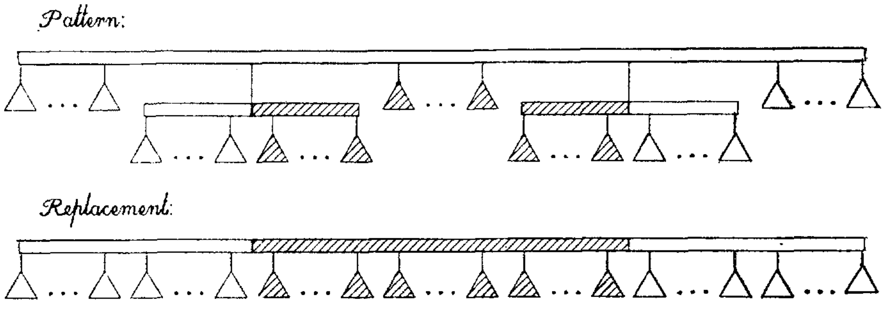

FIG. 15. Template $Q3$ for a doubly partial Q-node.

This is the last of the legal templates for Q-nodes. Any other pattern is illegal and results in a failure of the template-matching process. Using all of the templates specified above, each node within the tree is matched until the root of the pertinent subtree is found. Each node matches at most a single template. As long as legal templates are found the process continues. The matching stops when no more rearrangements are necessary; all of the pertinent leaves will be consecutive within the frontier.

As an illustration of the reduction algorithm consider the tree in Fig. 3. The set used to reduce the tree is $S = \{E, I, J, K\}$. The tree in Fig. 4 is equivalent to the tree in Fig. 3 and has all of the elements of S as a consecutive subsequence along its frontier. The algorithm should succeed. It does, and REDUCE(T, S) is the tree shown in Fig. 16. The two Q-nodes in the original tree have been merged into a single Q-node and one of the P-nodes has also been absorbed into the Q-node.

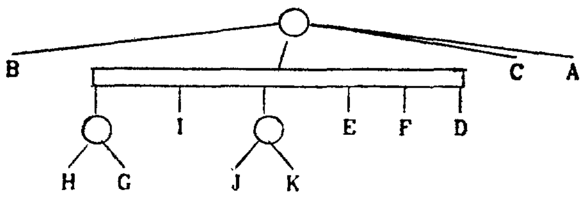

FIG. 16. A reduced PQ-tree.

Notice that the set S occurs as a consecutive subsequence within the frontier of REDUCE(T, S). Even better, it is consecutive within the frontier of every tree in the equivalence class of REDUCE(T, S). This is no accident. PQ-trees are designed to have this property. The S-reduction of a tree can be characterized in terms of consistent permutations. Given any subset $S \subset U$, let $T(U, S)$ be the PQ-tree shown in Fig. 17.

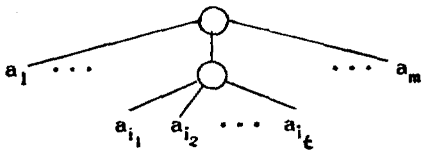

FIG. 17. $T(U, S)$ for $S = \{a_{i_1}, a_{i_2}, \ldots, a_{i_t}\}$.

Obviously CONSISTENT($T(U, S)$) is exactly the class of permutations in which the elements of S occur as 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. $T(\varnothing, \varnothing)$ 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 $S \subset U$ be any subset of the universal set.*

$$\begin{aligned}&\text{CONSISTENT(REDUCE}(T, S))\\&\quad = \text{CONSISTENT}(T) \cap \text{CONSISTENT}(T(U, S)).\end{aligned}$$

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 π be any permutation in the class CONSISTENT(REDUCE(T, S)). If such a permutation exists the tree must have been successfully reduced, otherwise REDUCE(T, S) would be the null tree. By definition the reduced form of T has an equivalent tree whose frontier is π. 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 π 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 left applied, 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 π, hence $\pi \in$ CONSISTENT(T).

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

(a) the descendants of X include all of S, and

(b) X 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 CONSISTENT(T) $\cap$ CONSISTENT($T(U, S)$) is contained in CONSISTENT(REDUCE(T, S)). That is, if a permutation π is in the consistent set of the original tree and happens to have the elements of S appearing consecutively, then π is also in the consistent set of the reduced tree. Let T' be a tree equivalent to T and having π as its frontier. The fact that S appears consecutively in π 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 π is in CONSISTENT(REDUCE(T', S)). Then since T' and T are equivalent, and since template matching preserves equivalence, π is also in

CONSISTENT-(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 an 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 requirement stems from the fact that the appropriate template for a node can only be selected after knowing the reduced forms of the node's children. The implementation discussed here scans the tree twice, both times beginning with the leaves and working toward the root. If that were all there were to the story, the algorithm would be quite easy.

But life is not so simple. Section 2 assumes that the entire tree is processed. Each node is classified as empty, full, or partial and then a pattern replacement is performed. But what if the sets being reduced are small? Scanning the entire tree each time would be an extremely expensive proposition.

The entire tree is not scanned. Not even all of the pertinent subtree is scanned. Instead, just a piece of the pertinent tree is checked. The ***pruned pertinent subtree of T with respect to S*** is the smallest connected subgraph (not necessarily a proper PQ-tree!) which contains all of the pertinent leaves. This is denoted by PRUNED(T, S). The root of the pruned pertinent subtree is ROOT(T, S), the root of the entire pertinent subtree. The pruned pertinent subtree for the pertinent subtree shown in Fig. 6 is shown in Fig. 18. The same set $S = \{E, I, J, K\}$ is used. In this case, the pruned pertinent subtree fails to satisfy the definition of a proper PQ-tree on at least two counts: it has a P-node with only one child and two Q-nodes with only two.

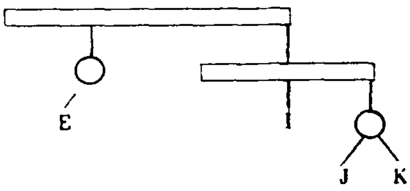

FIG. 18. The pruned pertinent subtree for the example in Fig. 6.

Two passes are used in the actual implementation. The first identifies the nodes to be processed and the second applies the templates. The combined algorithm is called ***pruned reduction*** because it only looks at the pruned pertinent subtree. The first pass is called ***bubbling up***. It marks all of the nodes which are in PRUNED(T, S). This is necessary in order to properly sequence the template-matching which follows. The two-pass strategy leads to an interesting dilemma.

Empty nodes are never actually looked at, so they must be recognized by their absence, rather than their presence. This is why the separate marking pass is used. As the marking bubbles up the tree from the pertinent leaves it is able to mark each node which needs to be processed and at the same time it leaves a count at each node telling exactly how many of the children will be processed. It is these counts which enable the second pass to determine when the last pertinent child of a node has been matched, so that the parent is properly queued. If this were not the case a parent with some empty children might never be queued.

The dilemma stems from the fact that a bottom-up strategy is employed only to avoid processing the entire tree but requires that each child point to its parent (how else can the algorithm bubble up?), yet the maintenance of parent pointers may easily require as much work as traversing the entire tree! Parent pointers cause a problem for one simple reason. If an internal node has a large number of empty children and the node is deleted as part of the replacement, every one of the children must receive a new parent pointer. It is possible that almost all of the nodes in the tree may receive a new parent even though the set S has only two elements.

The way out of the dilemma is to strike a balance between the desire to avoid scanning the entire tree and the need to minimize pointer upkeep. Parent pointers need only be kept for children of P-nodes and for endmost children of Q-nodes. Interior children do not need parent pointers. They can borrow them from their endmost siblings at the appropriate moment.

As the first pass bubbles up the tree it can be temporarily ***blocked*** at an interior node. If there is no parent pointer it is impossible to queue the parent. Each interior node without a parent pointer is called ***blocked.*** A count is kept of the number of times this happens. The actual count, kept as a global ***blockcount***, does not record all of the blocked nodes. Instead it keeps track of each ***block*** of blocked nodes. A block is a maximal left-to-right chain of blocked siblings. Thus two or more blocked nodes which are consecutive children of the same Q-node are only counted once. The difference between a blocked node and a block of blocked nodes is important, and should be kept in mind when thinking about the bubbling up pass.

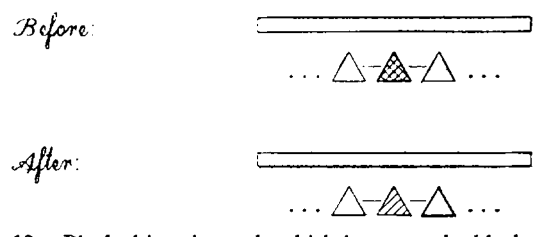

FIG. 19. Blocked interior node which increases the blockcount.

As occurred earlier for the template-matching, there are a number of cases. Children of P-nodes and endmost children of Q-nodes are easy to handle. The more complicated cases involve interior children of Q-nodes. Figure 19 depicts one situation when an interior node is encountered during the bubbling up phase. The node being processed is indicated by cross-hatching. Previously processed nodes are shaded. Parent pointers are only shown if they actually exist. The sibling pointers are always present and are shown in the diagrams. For the case shown here, there is no parent pointer, so the blockcount is increased.

It can also happen that both siblings have already been processed and that they are *unblocked.* 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.

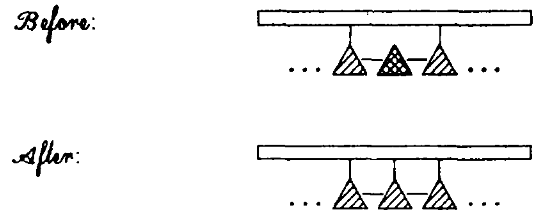

FIG. 20. Unblocked interior node leaving the blockcount unchanged.

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.

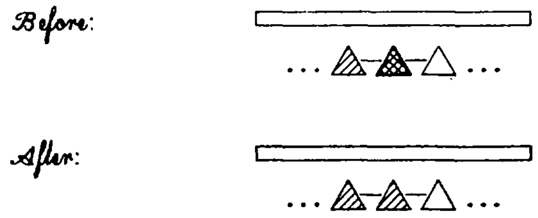

FIG. 21. Blocked interior node leaving the blockcount unchanged.

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.

A 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.

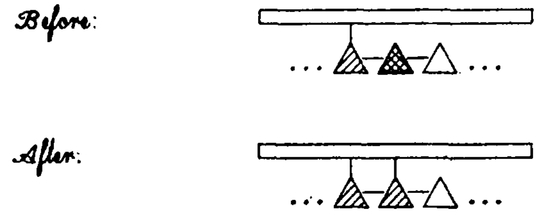

FIG. 22. Unblocked interior node leaving the blockcount unchanged.

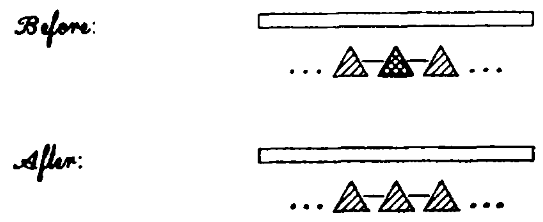

FIG. 23. Blocked interior node which decrements the blockcount.

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.

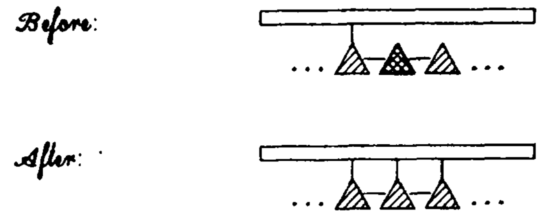

FIG. 24. Unblocked interior node which decrements the blockcount.

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 of one is legal. This is handled as a special case. The actual root is never needed, so a *pseudonode* can be assigned as a surrogate parent for the blocked children. The pseudonode can be removed afterwards.

If the blockcount is greater than one at the end of the bubbling up pass there is no hope that the tree can be reduced. The reduction algorithm can be halted immediately. There is only one more complication which remains to be covered. The procedure is supposed to halt after it has processed the pruned pertinent subtree. It should stop at the root of PRUNED(T, S). This is easier said than done. Instead the bubbling up continues until the blockcount is zero and the queue of nodes to process has only one node, or else the blockcount is one and the queue is empty. Either case is acceptable.

If the tree is actually irreducible the bubbling up may perform a lot of extra work. But at most it will process the entire tree. The applications for reduction always halt when the tree is irreducible, so this extra work does not really affect the total running time. The entire tree had to be built by the algorithm hence one additional pass over all of the nodes can not hurt!

If the bubbling up pass processes the root of the tree it must keep track of this fact. There are no higher nodes to process, but it must pretend that it is still adding nodes to the queue to ensure that proper end conditions are detected. Conceptually this can be remedied by imagining an infinite hierarchy of *virtual nodes* which are ancestors of the root of the tree. The actual implementation simply sets a flag to remember that the root of the tree has been processed. If the root of the entire tree is encountered and the queue empties with the blockcount nonzero, the tree is not reducible.

With a little care all of these details can be handled and the bubbling up procedure shown below will mark all of the pertinent nodes. It can also leave a count at each node of how many children are pertinent. This is used during the second pass to determine when the last child has been matched.

Before presenting the procedure for the first pass, a complete list of the global variables used by both passes is explained. The fields associated with each node are also helpful. The global variables are the following:

BLOCK_COUNT. The number of blocks of blocked nodes during the bubbling up pass.

BLOCKED_NODES. The number of blocked nodes during the bubbling up pass. This is only needed for the case when a pseudonode is used. The count at the end of the first pass is exactly the number of pertinent children for the pseudonode.

OFF_THE_TOP. A variable which is either 0 (its initial value) or 1 (if the root of the tree has been processed during the first pass). It acts as a count of the number of virtual nodes which are imagined to be in the QUEUE during the bubbling up.

QUEUE. A first-in first-out list which is used during both passes for sequencing the order in which nodes are processed.

Each node has the following fields. Not every field is used for every node so there is plenty of opportunity to reduce the storage requirement by reusing fields, but this possibility is ignored in the interest of clarity.

CHILD_COUNT. A count of the number of children currently possessed by the node. This is only used for *P*-nodes.

CIRCULAR_LINK. A set of links which form the children of a *P*-node into a doubly-linked circular list; the order of the list is arbitrary. The sole purpose of the circular list is to enable a *P*-node to find its only empty child when all other children are full or partial. This field is not used for children of *Q*-nodes.

ENDMOST_CHILDREN. A set containing the two endmost children of a *Q*-node. This is only used for *Q*-nodes.

FULL_CHILDREN. A set containing all of the children of a node which are currently known to be full. This is implemented as a list (no special order is necessary) and a count is kept of the number of nodes on the list. Children are added to the list after they are matched to a template in the second pass.

IMMEDIATE_SIBLINGS. A set containing exactly 0, 1, or 2 other nodes. A child of a *P*-node has no immediate siblings, the endmost children of *Q*-nodes have only one immediate sibling, and the interior children of *Q*-nodes have two immediate siblings. The field is interpreted as an unordered set. This convention allows sibling chains to be reversed without having to modify the pointers for every node of the chain. Sibling chains can be traversed in either direction beginning with any sibling in the chain.

LABEL. An indication of whether the node is empty, full, or partial.

MARK. A designation used during the first pass. Every node is initially *unmarked*. It is marked *queued* when it is placed onto QUEUE during the bubbling up. It is marked either ***blocked*** or ***unblocked*** when it is processed. Blocked nodes can become unblocked if their siblings become unblocked.

PARENT. The immediate ancestor of the node. This field is always valid for children of *P*-nodes and for endmost children of *Q*-nodes. It is only valid for interior children of *Q*-nodes if the child is marked as unblocked. The root of the *PQ*-tree is the unique node having no immediate siblings and no parent.

PARTIAL_CHILDREN. A set containing all of the children of a node which are currently known to be partial. This is similar to the set of full children. In practice it can have at most two elements, since otherwise the node will not match a legal template.

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 queued 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 $X \in S$ do place $X$ onto QUEUE;
  while | QUEUE | + BLOCK_COUNT + OFF_THE_TOP $> 1$ do
    begin
      if | QUEUE | $= 0$ then
        begin
          $T := T(\varnothing, \varnothing)$;
          exit from do
        end;
      remove $X$ from the front of QUEUE;
      MARK($X$) := "blocked";
      $BS$ := {$Y \in$ IMMEDIATE_SIBLINGS($X$) | MARK($Y$) = "blocked"};
      $US$ := {$Y \in$ IMMEDIATE_SIBLINGS($X$) | MARK($Y$) = "unblocked"};
      if | $US$ | $> 0$
        then
          begin
            choose any $Y \in US$;
            PARENT($X$) := PARENT($Y$);
            MARK($X$) := "unblocked"
          end
```

```
            else if | IMMEDIATE_SIBLINGS($X$)| $<$ 2 then MARK($X$) :=
             "unblocked";
        if MARK($X$) = "unblocked"
            then
                begin
                    $Y$ := PARENT($X$);
                    if | $BS$ | $>$ 0 then
                        begin
                            LIST := the maximal consecutive set of blocked siblings
                             adjacent to $X$;
                            for $Z \in$ LIST do
                                begin
                                    MARK($Z$) := "unblocked";
                                    PARENT($Z$) := $Y$;
                                    PERTINENT_CHILD_COUNT($Y$) :=
                                    PERTINENT_CHILD_COUNT($Y$) + 1
                                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
            else
                begin
                    BLOCK_COUNT := BLOCK_COUNT + 1 − | $BS$ |;
                    BLOCKED_NODES := BLOCKED_NODES + 1
                end
    end;
  return $T$
end
```

Note that LIST may be computed in time directly proportional to | LIST | through the use of the IMMEDIATE_SIBLINGS fields. Starting at any unblocked node X, we may traverse its chain of siblings in either direction. Siblings are added to LIST until one is encountered which is not blocked.

Note also that after BUBBLE all pertinent proper descendants of ROOT(T, S), except those which are blocked, have valid parent pointers. If BLOCK_COUNT is 1 and BLOCKED_NODES is greater than 1 when the BUBBLE procedure finishes, a pseudonode must be created. This process will be described only briefly here. During the BUBBLE procedure a list BLOCKED_LIST of all nodes which are ever marked "blocked" may be maintained. After the procedure finishes, all nodes which are no longer blocked are removed from BLOCKED_LIST. A pseudonode Z of type Q-node is created. Each node in BLOCKED_LIST is given Z as its parent. The ENDMOST_CHILDREN of Z are those nodes in BLOCKED_LIST which have fewer than 2 blocked siblings. Note that the only template which could match the true parent of the nodes in BLOCKED_LIST is $Q3$.

The bubbling up procedure never uses a parent pointer until after it is known to be valid. This eliminates the need to maintain pointers from all of the interior children but still enables BUBBLE to find all of the pertinent nodes with only a modest amount of work. The exact bound is given by the next lemma.

LEMMA 2. *The bubbling up phase of reduction requires* $O(|$PRUNED$(T, S)|)$ *steps.*

Proof. Note that the work performed during one iteration of the main loop is on the order of one plus the number of nodes which become unblocked during this iteration. Now each node is added to the queue at most once and becomes unblocked at most once. Thus the total work performed is bounded by the number of nodes processed. The nodes processed are those in PRUNED(T, S), plus some extra nodes which are ancestors of ROOT(T, S). There cannot be many of these extra nodes. The queueing strategy guarantees that if node X is processed before node X' then the parent of X must be processed before the parent of X', unless the parent of X' was already queued because of some other child being processed. Each ancestor of ROOT(T, S) has only one pertinent child. This implies that the total number of ancestors of ROOT(T, S) which are actually processed does not exceed the longest leaf-to-root distance within PRUNED(T, S). Certainly this is no more than | PRUNED(T, S)|. ▮

Bubbling up supplies all of the information necessary to perform the template matching. The second pass is essentially the procedure REDUCE which was explained in Section 2. The difference is that only the leaves which are in S are initially placed on the queue for processing. With some additional tricks the second pass can also be performed at modest cost. The final version of the template-matching is the procedure given below. Note that Template $P0$ and Template $Q0$ are not used.

Since only pertinent nodes are processed there are never any empty nodes in the queue. A new Template $L1$ has also been added for full leaves. The template's replacement is the same as its pattern—the leaf itself.

```
PQ-tree procedure REDUCE($T$, $S$);
begin
  initialize QUEUE to empty;
  for each leaf $X \in S$ do
    begin
      place $X$ on QUEUE;
      PERTINENT_LEAF_COUNT($X$) := 1
    end;
  while | QUEUE | > 0 do
    begin
      remove $X$ from the front of QUEUE;
      if | PERTINENT_LEAF_COUNT($X$)| < | $S$ |
        then
          begin comment $X$ is not ROOT($T$, $S$);
            $Y$ := PARENT($X$);
            PERTINENT_LEAF_COUNT($Y$) :=
             PERTINENT_LEAF_COUNT($Y$)
             + PERTINENT_LEAF_COUNT($X$);
            PERTINENT_CHILD_COUNT($Y$) :=
             PERTINENT_CHILD_COUNT($Y$) − 1;
            if PERTINENT_CHILD_COUNT($Y$) = 0 then
             place $Y$ onto QUEUE;
            if not TEMPLATE_$L1$($X$) then
            if not TEMPLATE_$P1$($X$) then
            if not TEMPLATE_$P3$($X$) then
            if not TEMPLATE_$P5$($X$) then
            if not TEMPLATE_$Q1$($X$) then
            if not TEMPLATE_$Q2$($X$) then
              begin
                $T$ := $T(\varnothing, \varnothing)$;
                exit from do
              end
          end
        else
          begin comment $X$ is ROOT($T$, $S$);
            if not TEMPLATE_$L1$($X$) then
            if not TEMPLATE_$P1$($X$) then
```

```
                if not TEMPLATE_$P2(X)$ then
                if not TEMPLATE_$P4(X)$ then
                if not TEMPLATE_$P6(X)$ then
                if not TEMPLATE_$Q1(X)$ then
                if not TEMPLATE_$Q2(X)$ then
                if not TEMPLATE_$Q3(X)$ then
                  $T := T(\varnothing, \varnothing)$;
                exit from do
              end
    end;
  return $T$
end
```

All that remains is to describe the templates. There are a few things to be careful of here also. Note that when Template $P5$ 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 $P5$ and $Q2$ 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_$P5(X)$;
begin
  if TYPE($X$) $\neq$ $P$-node then return false;
  if |PARTIAL_CHILDREN($X$)| $\neq 1$ then return false;
  $Y$ := the unique element in PARTIAL_CHILDREN($X$);
  $EC$ := the unique element in ENDMOST_CHILDREN($Y$) labeled "empty";
  $FC$ := 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 $ES$ := 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($X$);
  LABEL($Y$) := "partial";
  PARTIAL_CHILDREN(PARENT($Y$)) :=
    PARTIAL_CHILDREN(PARENT($Y$)) $\cup$ {$Y$};
  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 $X \in$ ENDMOST_CHILDREN(PARENT($Y$))
        then ENDMOST_CHILDREN(PARENT($Y$)) :=
          ENDMOST_CHILDREN(PARENT($Y$)) $- \{X\} \cup \{Y\}$
    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;
          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($FC$) := IMMEDIATE_SIBLINGS($FC$) $\cup \{ZF\}$;
    IMMEDIATE_SIBLINGS($ZF$) := $\{FC\}$;
    ENDMOST_CHILDREN($Y$) :=
      ENDMOST_CHILDREN($Y$) $- \{FC\} \cup \{ZF\}$
  end;
NUMBER_EMPTY := CHILD_COUNT($X$)
  $-$ | FULL_CHILDREN($X$)| $-$ | PARTIAL_CHILDREN($X$)|;
if NUMBER_EMPTY $>$ 0 then
  begin
    if NUMBER_EMPTY = 1
      then $ZE$ := $ES$
      else
```

```
            begin
              $ZE$ := $X$; LABEL ($ZE$):= "empty";
              CHILD_COUNT($ZE$) := NUMBER_EMPTY
            end;
        PARENT($ZE$) := $Y$;
        IMMEDIATE_SIBLINGS($EC$) := IMMEDIATE_SIBLINGS($EC$) ∪ {$ZE$};
        IMMEDIATE_SIBLINGS($ZE$) := {$EC$};
        ENDMOST_CHILDREN($Y$) :=
         ENDMOST_CHILDREN($Y$) − {$EC$} ∪ {$ZE$}
      end;
  if NUMBER_EMPTY $<$ 2 then destroy $X$;
  return true
end
Boolean procedure TEMPLATE_$Q$2($X$);
begin
  if TYPE($X$) $\neq$ $Q$-node then return false;
  if $X$ is a pseudonode then return false;
  if |PARTIAL_CHILDREN($X$)| $>$ 1 then return false;
  if | FULL_CHILDREN($X$)| $>$ 0
    then
      begin
        if | FULL_CHILDREN($X$) ∩ ENDMOST_CHILDREN($X$)| $\neq$ 1
          then return false;
        let $Y$ be the unique element in
         FULL_CHILDREN($X$) ∩ ENDMOST_CHILDREN($X$);
        for $i$ := 1 step 1 until | FULL_CHILDREN($X$)| do
          begin
            if $Y \in$ FULL_CHILDREN($X$) then return false;
            $Y$ := the next sibling in the chain of children of $X$
          end;
        if PARTIAL_CHILDREN($X$) $\not\subset$ {$Y$} then return false
      end
    else if PARTIAL_CHILDREN($X$) $\not\subset$ ENDMOST_CHILDREN($X$)
     then return false;
  LABEL($X$) := "partial";
  PARTIAL_CHILDREN(PARENT($X$)) :=
   PARTIAL_CHILDREN(PARENT($X$)) ∪ {$X$};
  if | PARTIAL_CHILDREN($X$)| $>$ 0 then
    begin
      $Y$ := the unique element of PARTIAL_CHILDREN($X$);
      $FC$ := the unique endmost full child of $Y$;
```

```
          if $Y$ has a full immediate sibling
            then
              begin
                let $FS$ be the unique full immediate sibling of $Y$;
                IMMEDIATE_SIBLINGS($FS$) :=
                  IMMEDIATE_SIBLINGS($FS$) $- \{Y\} \cup \{FC\}$;
                IMMEDIATE_SIBLINGS($FC$) :=
                  IMMEDIATE_SIBLINGS$\{FC\} \cup \{FS\}$
              end
            else
              begin
                ENDMOST_CHILDREN($X$) :=
                  ENDMOST_CHILDREN($X$) $- Y \cup \{FC\}$;
                PARENT($FC$) := $X$
              end;
          $EC$ := the unique endmost empty child of $Y$;
          if $Y$ has an empty immediate sibling
            then
              begin
                let $ES$ be the unique empty immediate sibling of $Y$;
                IMMEDIATE_SIBLINGS($ES$) :=
                  IMMEDIATE_SIBLINGS($ES$) $- \{Y\} \cup \{EC\}$;
                IMMEDIATE_SIBLINGS($EC$) :=
                  IMMEDIATE_SIBLINGS$\{EC\} \cup \{ES\}$
              end
            else
              begin
                ENDMOST_CHILDREN($X$) :=
                  ENDMOST_CHILDREN($X$) $- Y \cup \{EC\}$;
                PARENT($EC$) := $X$
              end;
          destroy $Y$
        end;
    return true
end
```

This completes the description of the template-matching. The next lemma shows that with the addition of a bubbling up phase the template-matching can be accomplished in much less time.

LEMMA 3. *The template-matching phase of reduction requires* $O(|\text{PRUNED}(T, S)|)$ *steps.*

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 ***pruned reduction algorithm*** is simply BUBBLE followed by REDUCE. The overall algorithm is as follows:

```
PQ-tree procedure REDUCTION($U$, $\mathbb{S}$);
  $T := T(U, U)$;
  for each $S \in \mathbb{S}$ do
    begin
      $T :=$ BUBBLE($T$, $S$);
      $T :=$ 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 UNARY(T, S) be the set of all unary nodes in T with respect to the set S.

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

Proof. There are only $|S|$ leaves. Binary branching implies that there are at most $O(|S|)$ nonunary nodes in 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 $\mathbb{S}$ define SIZE($\mathbb{S}$) 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 $\mathbb{S}$ occurs as a consecutive subsequence can be computed in* $O(m + n + \text{SISE}(\mathbb{S}))$ *steps if U has m objects and $\mathbb{S}$ 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 SIZE($\mathbb{S}$). The second term is the sum of $|$ UNARY$(T, S)|$ over all sets S in $\mathbb{S}$. This term is a bit harder to bound. We begin by noting that a unary node cannot be the root of PERTINENT(T, S) and that its children cannot be all empty or all full. Thus the only templates which can apply to unary nodes are $P3$, $P5$, and $Q2$. First consider Template $P3$. 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 $Q2'$ be Template $Q2$ when no partial children are present and let $Q2''$ be Template $Q2$ when one partial child is present. By an argument like that used for $P3$, we see that $Q2'$ can be applied a total of $O(m + n)$ times. Now let 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, NORM(T) is at most m.

(b) No template replacement increases NORM(T) by more than one.

(c) Templates $P5$ and $Q2''$ reduce NORM(T) by at least one.

Now we have already seen that the number of applications of all templates except $P5$ and $Q2''$ is $O(m + n + \text{SIZE}(\mathbb{S}))$. Then since NORM$(T)$ is clearly nonnegative, (a), (b), and (c) imply that the number of applications of $P5$ and $Q2''$ is $O(m + n + \text{SIZE}(\mathbb{S}))$. 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* iff 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].

$$\begin{pmatrix} 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 0 \end{pmatrix} \qquad \begin{pmatrix} 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 \end{pmatrix} \qquad \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

Consecutive ones **Circular ones** **Neither property**

$$\begin{pmatrix} 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix} \qquad \begin{pmatrix} 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 \end{pmatrix} \qquad ?$$

FIG. 25. Matrices with consecutive and circular ones.

The matrix on the top left in Fig. 25 has the consecutive ones property for columns. The original version is not in consecutive form but the rows can be rearranged as shown in the bottom left matrix so that the ones are consecutive. The definition merely requires that *some* ordering of the rows places the ones into consecutive form, hence the matrix is deemed to have the consecutive ones property. The top center matrix has the circular ones property because the rows can be rearranged as shown on the bottom center so that all of the columns have circular ones. There is no rearrangement of the rows which will place it into consecutive form so it does not have the consecutive ones property. The matrix on the top right has neither a circular form nor a consecutive form; there is no arrangement of the rows which achieves either circular or consecutive ones.

Matrices having consecutive ones and other related properties occur in many fields, including archeology [16], information retrieval [10], and table-driven parsing [15]. Tucker [31] has given a structure theorem for the consecutive ones property. The theorem is similar to Kuratowski's [17] forbidden subgraph characterization of planar graphs.

Fulkerson and Gross published an $O(mn^2)$ algorithm for testing an $m \times n$ matrix for the consecutive ones property [8]. This bound can be improved by noticing that the dominant term is due to the computation of a matrix product. Using the techniques developed by Strassen [28], the Fulkerson–Gross algorithm can be implemented to run in $O(\max\{m^{(\log_2 7)-2}n^2, mn^{(\log_2 7)-1}\})$ steps for an $m \times n$ (0, 1)-matrix [3]. Using the reduction algorithm an even better bound can be obtained. The basic algorithm is the following procedure.

```
Boolean procedure CONSECUTIVE($M$);
begin
  let $U$ be the set of rows in $M$;
  $T := T(U, U)$;
  **for** $j := 1$ **step** 1 **until** $n$ **do**
    **begin**
      let $S$ be the set of rows having a one in column $j$;
      $T :=$ BUBBLE($T, S$);
      $T :=$ REDUCE($T, S$);
      **if** $T = T(\varnothing, \varnothing)$ **then return false**
    **end**;
  **return true**
**end**
```

THEOREM 6. *If M is an $m \times n$ (0, 1)-matrix specified by its f nonzero entries a consecutive ones test can be performed in $O(m + n + f)$ steps.*

Proof. The algorithm shown above performs the desired test. The number of elements in the universal set is m, the number of sets is n, and the total size of the sets is clearly f. All of the work except for the reduction steps is clearly linear. Using the results of Theorem 5 it is easy to show that the reductions require $O(m + n + f)$ steps. The entire algorithm is thus linear. ▌

A similar result holds for the circular ones property. Tucker observed the following interesting construction [30]. Given a (0, 1)-matrix M choose any row of the matrix. Let M^c be the matrix obtained from M by complementing every column of M which has a one in the selected row.

LEMMA 7 (Tucker). *A (0, 1)-matrix M has the circular ones property for columns iff M^c has the consecutive ones property for columns.*

Proof. The key observation is that after the complementation the new matrix M^c has zeros everywhere in the selected row. Without loss of generality it can be assumed that the permutation of the rows which places M into circular form does so by placing the selected row at the top of the matrix. Such a permutation must then place M^c into consecutive form. The truly circular case is ruled out, so testing M^c for consecutive ones is sufficient. Recomplementing the consecutive form of M^c yields a circular form for M. ▌

Tucker's construction does not guarantee a fast test unless a judicious selection is made for the row which determines the complementation. An all-ones row would result in having to complement the entire matrix. This is easily avoided.

LEMMA 8. *If M is an $m \times n$ (0, 1)-matrix having f nonzero entries there exists an $m \times n$ (0, 1)-matrix M^c having at most $2f$ nonzero entries such that M has the circular* *ones property iff M^c has the consecutive ones property. M^c 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^c 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^c 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^c is within the desired time bound. The number of edges in M^c is $f' \leqslant 2f$ 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. Hajós 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 PQ-trees this can be lowered to $O(n + e)$ for graphs with n vertices and e 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.

The dominant clique vs vertex matrix for G has a row for each dominant clique and a column for each vertex, with an entry being nonzero iff the vertex is a member of the clique.

THEOREM 10 (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 so that a list of nonzero entries can be constructed 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 in which for 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 $O(n^{\log_2 7})$ 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 versus vertex matrix can be constructed in $O(n + e)$ steps using this technique. A useful property of chordal graphs is the fact that the dominant-clique vs vertex matrix never has more than $O(n + e)$ 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** $v \in V$ **do**
    **begin**
      let $S$ be the set of cliques containing $v$;
      $T :=$ BUBBLE($T$, $S$);
      $T :=$ REDUCE($T$, $S$);
      **if** $T = T(\varnothing, \varnothing)$ **then return false**
    **end**;
  **return true**
**end**
```

THEOREM 11. *A graph $G = (V, E)$ can be tested for being an interval graph in $O(n + e)$ steps when G has n vertices and e edges.*

Proof. The output of the $O(n + e)$ chordality test can be used as the input to the consecutive ones test. The total number of nonzero entries is $O(n + e)$ so the overall bound is still $O(n + e)$. ▌

It is interesting to note that the consecutive ones test required during the interval graph recognition algorithm can be simplified somewhat by processing the vertices of the graph in a special order. This ordering is obtained by using a modified form of the lexicographic breadth first search mentioned above. For details see [21].

A by-product of the interval graph test is the reduced PQ-tree whose leaves are the dominant cliques. It is possible to use this tree as the basis for a canonical representation for interval graphs. Using an algorithm similar to the tree isomorphism algorithm in [1], one may test for isomorphism of interval graphs in $O(n + e)$ time. This is discussed in greater detail in [21].

The importance of this last result is increased by the observation that the existence of a polynomial-time algorithm for solving general graph isomorphism is equivalent to the existence of a polynomial-time isomorphism test for either *chordal* or *transitively-orientable* [11] graphs. (All interval graphs are chordal and have transitively orientable complements.) Hirschberg and Edelberg [12] proved this equivalence for bipartite graphs. The bipartite result implies the result for transitively-orientable graphs. The proof for chordal graphs is similar and is given in [21].

6. PLANARITY

PQ-trees can be applied to problems other than those which arise directly from the consecutive ones property. The PQ-tree reduction algorithm can be used to build an efficient test for planar graphs. Hopcroft and Tarjan have already presented a linear test for planarity [14]. This section discusses a "new" algorithm for testing graph planarity which also runs in linear time. The algorithm is of interest for two reasons. It is a distinctly different application for PQ-trees and it improves an existing algorithm by introducing the appropriate data structure.

Lempel, Even, and Cederbaum [20] have published an algorithm for testing planarity which is based on formula manipulation. The algorithm employs a reduction operation almost identical to the one used here. Their version uses formulas to represent a graph and does not have a linear time bound. A linear version of the algorithm has been implemented, but it does not perform the formula manipulation [6]. The version here achieves the linear bound while implementing the original formula manipulation. The trick is to represent formulas by PQ-trees.

The possibility of using PQ-trees for planarity testing was suggested by Tarjan

[29] after *PQ*-trees had been devised for the consecutive ones problem. He noticed that the reduction operation for *PQ*-trees is very similar to the formula manipulation carried out by the Lempel, Even, and Cederbaum algorithm.

Before explaining the planarity algorithm, two facts about graphs are needed as background. The first is well known from graph theory, the second is proven in [20].

LEMMA 12. *A graph G is planar iff each of its biconnected components is planar.*

LEMMA 13 (Lempel–Even–Cederbaum). *If $G = (V, E)$ is a biconnected graph then its vertices can be numbered so that* 1 *and n are adjacent and, for any vertex numbered $1 < j < n$, there exist vertices numbered i and k such that $i < j < k$ and both i and k are adjacent to j.*

Algorithms requiring $O(n + e)$ steps for graphs having n vertices and e edges exist for finding biconnected components [1] and for generating the desired numbering [7]. It is assumed that all graphs to be considered are biconnected and that the vertices are numbered as in Lemma 13. The graph with its vertices numbered as indicated in Fig. 26 will be used throughout this section as an example of the planarity algorithm.

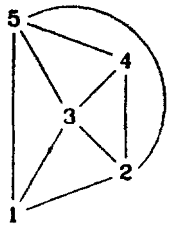

FIG. 26. A numbered, biconnected graph.

The algorithm tests a graph $G = (V, E)$ having n vertices and e edges. The algorithm assumes that $n > 2$ since otherwise G is trivially planar. Edges in E are considered to be directed from the lower numbered vertex to the higher. Edges can be directed at the same time the numbering is generated. A *PQ*-tree is used to represent the graph. The elements of the universal set are edges, although U will change during the course of the algorithm. An edge is added to U during the iteration corresponding to its lower-numbered vertex and it is removed from U during the iteration corresponding to its higher-numbered vertex.

```
**Boolean procedure** PLANAR($V$, $E$);
**begin**
    $U$ := the set of edges whose lower-numbered vertex is 1;
    $T$ := $T(U, U)$;
    **for** $j$ := 2 **step** 1 **until** $n - 1$ **do**
    begin
      $S :=$ the set of edges whose higher-numbered vertex is $j$;
      $T :=$ BUBBLE($T, S$);
      $T :=$ REDUCE($T, S$);
      if $T = T(\varnothing, \varnothing)$ then return false;
      $S' :=$ the set of edges whose lower-numbered vertex is $j$;
      if ROOT($T, S$) is a $Q$-node
        then replace the full children of ROOT($T, S$) and their
          descendants by $T(S', S')$
        else replace ROOT($T, S$) and its descendants by $T(S', S')$;
      $U := U - S \cup S'$
    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 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.

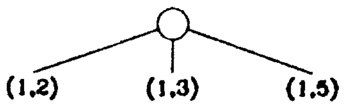

FIG. 27. Initial PQ-tree for planarity testing.

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.

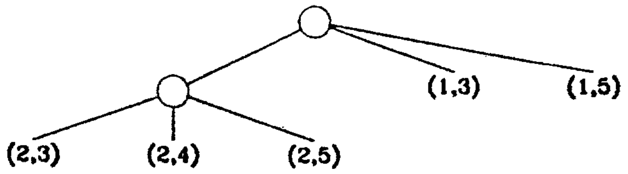

FIG. 28. *PQ*-tree after first iteration of inner loop.

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.

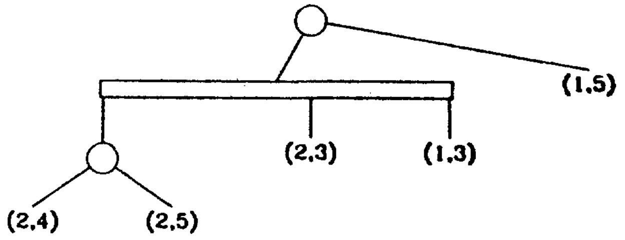

FIG. 29. *PQ*-tree after reducing edges entering vertex 3.

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.

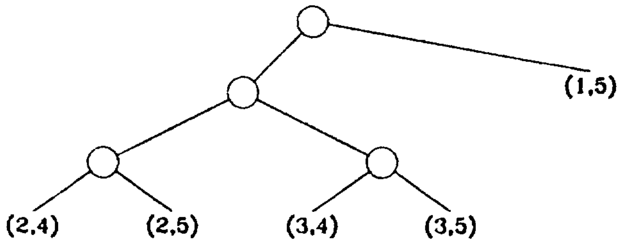

FIG. 30. *PQ*-tree after substituting edges leaving vertex 3.

For the final iteration, $j = 4$, the reduction leaves almost no freedom to the edges. All but one of the edges are forced to have a rigid left-to-right order. This is shown in Fig. 31.

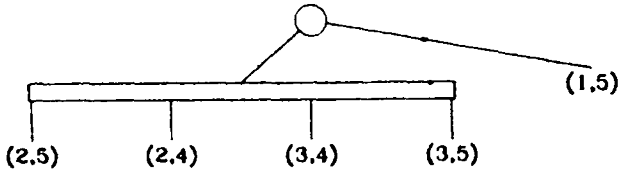

FIG. 31. PQ-tree after reduction when $j = 4$.

Substituting the single edge (4, 5) for the edges (2, 4) and (3, 4) produces the final form of the PQ-tree, shown in Fig. 32.

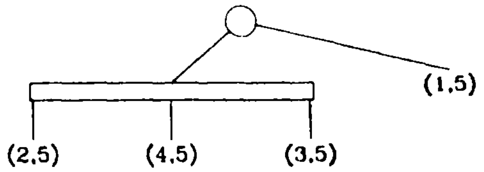

FIG. 32. Final PQ-tree for planarity test.

The algorithm halts with the value **true**, surely the correct response in light of the planar representation initially given in Fig. 26. The graph tested is a familiar one. Adding the other possible edge produces K_5, one of the two forbidden subgraphs of Kuratowski's structure theorem for planar graphs [17]. K_5 is known to be non-planar. Rerunning the algorithm with this added edge will indeed produce the answer **false**.

THEOREM 14. *Given a graph G having n vertices the planarity algorithm requires at most $O(n)$ steps.*

Proof. As mentioned earlier, $O(n + e)$ algorithms exist to find biconnected components and the numbering of Lemma 13, so we assume G is biconnected and numbered. Outside of the calls to BUBBLE and REDUCE, the algorithm can easily be seen to require only $O(n + e)$ time.

To see that the calls to REDUCE and BUBBLE use only $O(n + e)$ time, we use an argument just like that in Theorem 5. The only significant difference here is the fact that parts of the tree are replaced by $T(S', S')$. It is easy to see that this replacement can increase NORM(T) by at most $| S' |$. But the sum of $| S' |$ over all iterations is $O(n + e)$, so we again obtain an overall time bound of $O(n + e)$ for the calls to BUBBLE and REDUCE.

So far we have seen that the algorithm requires $O(n + e)$ time. To obtain a bound of $O(n)$, we use the trick employed in [14]: since a planar graph on n vertices has at most $3n - 3$ edges, we may immediately reject inputs with more than $3n - 3$ edges. For the remaining graphs, $O(n + e)$ is the same as $O(n)$. ▌

7. Conclusion

A new data structure called a *PQ*-tree has been introduced. *PQ*-trees are used for representing the classes of permutations within which various subsets of a universal set appear as consecutive subsequences. These permutations are useful for solving three problems: testing for the consecutive ones property, recognizing interval graphs, and testing graph planarity. For these problems algorithms based on *PQ*-trees are linear in the size of their input when implemented on a random access computer.

The consecutive ones test is an improvement over previously published algorithms. Interval graph recognition uses the consecutive ones test and a test for graph chordality. The linear time bound is again an improvement over previously published algorithms. The planarity test is a speeded-up version of an existing algorithm, due to Lempel, Even, and Cederbaum. For planarity the linear time bound is not new. The algorithm of Hopcroft and Tarjan is also linear. It would be of interest to compare actual implementations of these two planarity algorithms to estimate the constants of proportionality and to analyze the average behavior instead of the asymptotic worst-case behavior. This has not been done, and is suggested as a topic for further study.

Additional material and references may be found in both [3] and [21]. Many of the references were originally cited in [22].

Acknowledgments

The authors wish to thank Professors Richard M. Karp, Robert E. Tarjan, and Jeffrey D. Ullman for their many helpful suggestions during both the research and presentation of these results. John Beatty, Shimon Even, Michael Fischer, Richard Ladner, and Jan van Leeuwen have all read drafts of this article and each has pointed out numerous opportunities to eliminate errors or improve clarity.

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. S. Benzer, On the topology of the genetic fine structure, *Proc. Nat. Acad. Sci. U.S.A.* **45** (1959), 1607–1620.

3. 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.)
4. E. G. Coffman, Jr. and R. L. Graham, Optimal scheduling for two-processor systems, *Acta Informatica* **1** (1972), 200–213.
5. G. A. Dirac, On rigid circuit graphs, *Abh. Math. Sem. Univ. Hamburg* **25** (1961), 71–76.
6. S. Even, personal communication.
7. S. Even and R. E. Tarjan, Computing an st-numbering, *Theoretical Computer Science*, to appear.
8. D. R. Fulkerson and O. A. Gross, Incidence matrices and interval graphs, *Pacific J. Math.* **15** (1965), 835–855.
9. F. Gavril, An algorithm for testing chordality of graphs, *Inform. Proc. Lett.* **3**, No. 4 (1975), 110–112.
10. S. P. Ghosh, File organization: the consecutive retrieval property, *Comm. ACM* **9** (1972), 802–808.
11. P. C. Gilmore and A. J. Hoffman, A characterization of comparability graphs and of interval graphs, *Canad. J. Math.* **16** (1964), 539–548.
12. D. Hirschberg and M. Edelberg, On the complexity of computing graph isomorphism, Technical Report TR-130, Princeton University, Computer Science Laboratory, Department of Electrical Engineering, Princeton University, Princeton, N.J., August 1973.
13. G. Hajós, Über eine art von graphen, *Internationale Mathematische Nachrichten* **11** (1957), 65.
14. J. E. Hopcroft and R. E. Tarjan, Efficient planarity testing, *J. ACM* **21** (1974), 549–568.
15. G. Jennings, Representation of a collection of finite sets as intervals on a line, manuscript, Courant Institute, New York University, 1974.
16. D. G. Kendall, Incidence matrices, interval graphs and seriation in archaeology, *Pacific J. Math.* **28**, No. 3 (1969), 565–570.
17. K. Kuratowski, Sur le problème des courbes gauches en topologie, *Fund. Math.* **15** (1930), 271–283.
18. R. E. Ladner, M. J. Fischer, and S. Young, private communication.
19. 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), 45–64.
20. 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.), pp. 215–232, Gordon and Breach, New York, 1967.
21. G. S. Lueker, Efficient algorithms for chordal graphs and interval graphs, Ph. D. Dissertation, Program in Applied Mathematics and the Department of Electrical Engineering, Princeton University, Princeton, N.J., 1975.
22. F. S. Roberts, "Discrete Mathematical Models, with Applications to Social, Biological, and Environmental Problems," Prentice–Hall, Englewood Cliffs, N.J., to appear.
23. D. Rose, Triangulated graphs and the elimination process, *J. Math. Anal. Appl.* **32** (1970), 597–609.
24. D. J. Rose, R. E. Tarjan, and G. S. Lueker, Algorithmic aspects of vertex elimination on graphs, *SIAM J. Computing* **5**, No. 2 (June 1976), 266–283.
25. H. J. Ryser, Combinatorial configurations, *SIAM J. Appl. Math.* **17**, No. 3 (May 1969), 593–602.
26. R. Sethi, Scheduling graphs on two processors, *SIAM J. Computing* **5**, No. 1 (March 1976), 73–82.
27. K. E. Stoffers, Scheduling of traffic lights—a new approach, *Transportation Research* **2** (1968), 199–234.

28. V. STRASSEN, Gaussian elimination is not optimal, *Numer. Math.* **13** (1969), 354–356.
29. R. E. TARJAN, personal communication.
30. A. C. TUCKER, Matrix characterizations of circular-arc graphs, *Pacific J. Math.* **39**, No. 2 (1971), 535–545.
31. A. C. TUCKER, A structure theorem for the consecutive 1's property, *J. Combinatorial Theory* **12(B)** (1972), 153–162.