PC-Trees vs. *PQ*-Trees

Wen-Lian Hsu

Institute of Information Science, Academia Sinica, Taipei
hsu@iis.sinica.edu.tw,
http://www.iis.sinica.edu.tw/IASL/hsu/eindex.html

Abstract. A data structure called *PC*-tree is introduced as a generalization of *PQ*-trees. *PC*-trees were originally introduced in a planarity test of Shih and Hsu where they represent partial embeddings of planar graphs. *PQ*-trees were invented by Booth and Lueker to test the consecutive ones property in matrices. The original implementation of the *PQ*-tree algorithms by Booth and Lueker using nine templates in each bottom-up iteration is rather complicated. Also the complexity analysis is rather intricate. We give a very simple *PC*-tree algorithm with the following advantages: (1) it does not use any template; (2) it does all necessary operations at each iteration in one batch and does not involve the cumbersome bottom-up operation. *PC*-trees can be used naturally to test the circular ones property in matrices. And the induced *PQ*-tree algorithm can considerably simplify Booth and Lueker's modification of Lempel, Even and Cederbaum's planarity test.

1. Introduction

A data structure called *PC*-tree is considered here as a generalization of *PQ*-trees. *PC*-trees were originally introduced to represent partial embeddings of planar graphs in Shih and Hsu [7]. *PQ*-trees were used to test the consecutive ones property in matrices [1]. However, the implementation of *PQ*-tree algorithms by Booth and Lueker [1] (hereafter, referred to as the *B&L algorithm*) is rather complicated. Also the complexity analysis is rather intricate. We shall present a very simple *PC*-tree algorithm without using any template. Furthermore, we shall illustrate how to test the circular ones property in matrices using *PC*-trees.

PQ-trees were invented for the more general purpose of representing all permutations of a set U that are consistent with constraints of consecutiveness given by a collection C of subsets of U with the convention that the element of each subset S in C must occur consecutively in the permutation.

The literature on problems related to *PQ*-trees is quite extensive. *PQ*-trees can be applied to test the consecutive ones property of (0,1)-matrices [1,3], to recognize interval graphs [1,2] and to recognize planar graphs efficiently [1,6]. Korte and Möhring [5] considered a modified *PQ*-tree and a simpler incremental update of the tree for the purpose of recognizing interval graphs. Klein and Reif [4] constructed efficient parallel algorithms for manipulating *PQ*-trees. On the other hand, PC-trees were initially used in Shih and Hsu [7] (*S&H*) to represent partial embeddings of planar graphs. Note that this approach is entirely different from Lempel, Even and

J. Wang (Ed.): COCOON 2001, LNCS 2108, pp. 207–217, 2001.

Cederbaum's (*LEC*) approach [6] of using *PQ*-trees to test the consecutive ones property of all nodes adjacent to the incoming node in their vertex addition algorithm. In *S&H*, a P-node is a regular node of the graph, a C-node represents a biconnected component and nodes adjacent to the node in consideration can be scattered anywhere around the *PC*-tree. However, in *LEC*, the leaves of the *PQ*-tree must be those nodes adjacent to the incoming node.

In this paper we shall focus on the application of *PQ*-trees to (0,1)-matrices. A (0,1)-matrix *M* has the *consecutive ones property* (*COP*) for the rows iff its columns can be permuted so that the ones in each row are consecutive. *M* is said to have the *circular ones property* (*CROP*) for the rows if either the ones or the zeros of each row are consecutive. When the zeros of a row are consecutive the ones will wrap around the first and the last column.

A *PQ*-tree is a rooted tree *T* with two types of internal nodes: *P* and *Q*, which will be represented by circles and rectangles, respectively. The leaves of *T* correspond 1-1 with the columns of *M*.

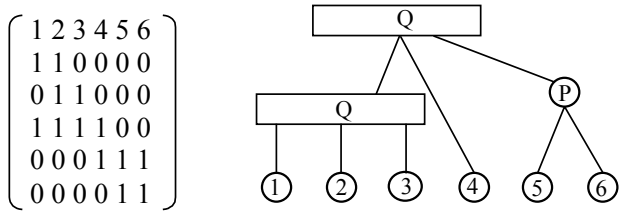


Fig. 1. A *PQ*-tree and the consecutive ones property

We shall follow the notations used in [1]. The frontier of a *PQ*-tree *T*, denoted by *FRONTIER*(*T*), is the permutation of the columns obtained by ordering the leaves of *T* from left to right. Such a permutation is called a *consistent permutation*. The *B&L* algorithm considers each row as a constraint and adds them one at a time. Each time a new row is added, the algorithm tries to modify the current *PQ*-tree to satisfy the consecutiveness constraint of columns of the newly added row. To guarantee unique *PQ*-tree representations we need to restrict ourselves to *proper PQ*-trees defined in [1]: every *P*-node has at least two children; every *Q*-node has at least three children. Two *PQ*-trees are *equivalent* iff one can be transformed into the other by applying zero or more equivalent transformations. The equivalence of two trees is written $T \sim T'$. There are two types of equivalent transformations:

1. Arbitrarily permute the children of a *P*-node,
2. Reverse the order of the children of a *Q*-node.

Denote the set of all consistent permutations (or frontiers) of a *PQ*-tree *T* by $CONSISTENT(T) = \{FRONTIER(T') \mid T' \sim T\}$. Given a *PQ*-tree *T* and a row *u* of *M*, define a new tree called the *u-reduction* of *T* to be one whose consistent permutations are exactly the original permutations in which the leaves in *u* occur consecutively. This new tree is denoted by *REDUCE*(*T*,*u*). Booth and Lueker gave a procedure *REDUCE* to obtain the *u*-reduction. The procedure applies a sequence of *templates* to the nodes of *T*. 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. This is a bottom-up strategy that examines the tree node-by-node obeying the child-before-parent discipline.

A node is said to be *full* if all of its descendants are in u; It is *empty* if none of its descendants are in u; If some but not all of the descendants are in u, it 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 u, denoted by $PERTINENT(T, u)$, is the subtree of minimum height whose frontier contains all columns in u. The root of the pertinent subtree is denoted by $ROOT(T, u)$, which is usually not the root of the entire tree. There are nine templates used in [1].

In order to achieve optimal efficiency, the algorithm takes several precautions in scanning the pertinent subtree. For example, the maintenance of parent pointers may cause a problem. It is possible that almost all nodes in the tree may receive a new parent even though the row u has only two elements. Therefore, parent pointers are only kept for children of *P*-nodes and for endmost children of *Q*-nodes. Interior children of a *Q*-node do not keep parent pointers. Rather, they borrow them from their endmost siblings. These measures unavoidably complicate the implementation of the B&L algorithm.

A *PC*-tree is a rooted tree T with two types of internal nodes: *P* and *C*, which will be represented by circles and double circles, respectively. The leaves of T correspond 1-1 with the columns of M. Figure 2 gives an example of a matrix satisfying the *CROP* (but not the *COP*) and its corresponding *PC*-tree. In obtaining consistent permutations, the children of a *P*-node can be permuted arbitrarily whereas the children of a *C*-node observe a circular list, whose order can only be changed from clockwise to counter-clockwise. The $FRONTIER(T)$ of a *PC*-tree consists of those *consistent circular permutations*.

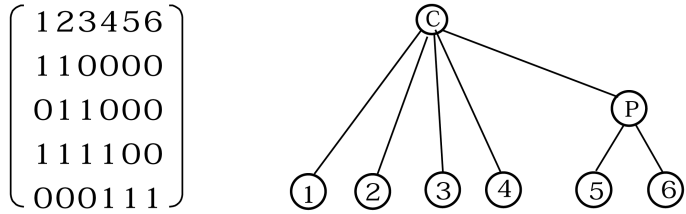


Fig. 2. A *PC*-tree and the circular ones property

For each *PQ*-tree, we can obtain its corresponding *PC*-tree by replacing the *Q*-nodes with *C*-nodes. Note that flipping a *Q*-node in a *PQ*-tree is the same as changing the circular children list of its corresponding *C*-node from clockwise to counter-clockwise order. There are a few differences between *PC*-trees and *PQ*-trees: (1) When a *PQ*-tree is transformed into a *PC*-tree, and its root is a *Q*-node, then its two endmost children must be preserved in order to maintain the *COP* in any consistent circular permutation of the transformed *PC*-tree. Aside from this restriction, all other *PC*-tree operations will yield equivalent *PQ*-tree operations. (2) Since it is the circular order of the leaves that needs to be preserved (for example, in Figure 2, we can rotate the tree so that *FRONTIER* becomes 2,3,4,5,6,1), one might as well fix the first column to be 1 in a *PC*-tree in considering any consistent circular permutation. (3)

The root of a *PC*-tree in general is not essential except to maintain the child-parent relationships.

With the additional freedom of rotation, it is natural for *PC*-trees to test the *CROP*. Since our purpose is to illustrate how to replace the complicated bottom-up template matching strategy in *B&L* algorithm, we shall concentrate on testing the *COP* using *PC*-trees.

Our improvement over the B&L algorithm is based on a simple observation that there exist at most two special partial nodes and the unique path connecting them gives a streamline view of the update operation. In particular, no template is necessary in our *PC*-tree algorithm and the node-by-node examination is replaced by *one* swift batch operation.

2. A Forbidden Structure of Consistent Permutations

A key observation that simplifies our *PC*-tree algorithm is based on the following forbidden structure of consistent permutations.

Theorem 1. The following structure $\{S_1, S_2, S_3\}$ is forbidden in any consistent permutation of the columns of M that preserves the *COP*.
Let $S_1 = \{a_1,b_1\}$, $S_2 = \{a_2,b_2\}$ and $S_3 = \{a_3,b_3\}$ be three subsets of columns such that

(1) Columns a_1, a_2 and a_3 are distinct from each other.

(2) None of the a_i's is the same as any of the bi's (but b_1, b_2 and b_3 are not necessarily distinct).

(3) No two columns in any S_i are separated by any column in the other two subsets. Furthermore, b_1, b_2 and b_3 are not separated by any of the a_i's.

Proof. If $b_1 = b_2$, then from (3), we must have the arrangement $a_1 \ldots b_1 (= b_2) \ldots a_2 \ldots$ But then, it would be impossible to place b_3 anywhere in the arrangement unless $b_1 = b_2 = b_3$. In the latter case it would be impossible to place a_3 anywhere in the arrangement.

Hence, assume $b_1 \neq b_2$. From (3), we must have the following arrangement $a_1 \ldots b_1 \ldots b_2 \ldots a_2 \ldots$ (or the reverse) in any consistent permutation of the columns. Since b_1, b_2 and b_3 must not be separated by the a_i's, we shall have $a_1 \ldots b_1 \ldots b_3 \ldots b_2 \ldots a_2 \ldots$ Note that in the above arrangement, b_3 is allowed to be the same as either b_1 or b_2. But then, it would be impossible to place a_3 anywhere in the arrangement.

3. Terminal Nodes and Paths

Define a partial node to be a *terminal node* if none of its children is partial. In other words, each child of a terminal node is either empty or full. Terminal nodes play a major role in simplifying our tree-update operation.

Theorem 2. If the given matrix M satisfies the *COP*, then there can be at most two terminal nodes in T at every iteration.

Proof. Suppose, on the contrary, there are three terminal nodes t_1, t_2, and t_3. Choose an empty child a_i and a full child b_i from each of the t_i, $i = 1, 2, 3$. Then, $S_1 = \{a_1,b_1\}$, $S_2 = \{a_2,b_2\}$ and $S_3 = \{a_3,b_3\}$ constitute a forbidden structure as described in Theorem 1.

If there are two terminal nodes t_1 and t_2, define the unique path connecting t_1 and t_2 to be the *terminal path*. In case there is only one terminal node t, the terminal path is defined to be the unique path from t to $ROOT(T,u)$. In the latter case, t is called a *special terminal node* if it is a *C*-node and its two endmost children are empty. That is, its full children are consecutively arranged in the middle.

Now, any *C*-node in the interior of the terminal path has its children (not on the path) divided into two sides. Since one can flip the children ordering along the terminal path to obtain an equivalent *PC*-tree, we shall show that there is a unique way to flip the children of these *C*-nodes "correctly". Denote $ROOT(T,u)$ by m.

Theorem 3. If M satisfies the *COP* and there are two terminal nodes t_1 and t_2, then
(1) Every *C*-node in the interior of the terminal path satisfies that its full children together with its two neighbors on the path are consecutive in its circular list. Therefore, they can be flipped correctly.
(2) Node m must be on the terminal path.
(3) Let w be a child of a node (other than t) on the unique path from m to the root of T such that w itself is not on this path. Then w must be empty.
Proof. Consider the following cases:

(1) If a *C*-node w does not have any full child, we are done. Hence, assume w has a full child d. Let the two neighbors of w in the terminal path be s_1 and s_2. Suppose, in traversing the circular list of w from s_1 to s_2 through the full child d_1, we encounter an empty child d_2, say in the order s_1, d_1, d_2, s_2. Let d_1', d_2' be two leaves of d_1, d_2, respectively. Choose an empty child a_i and a full child bi from each of the t_i, $i = 1, 2$.

Then $S_2 = \{a_1,b_1\}$, $S_2 = \{a_2,b_2\}$ and $S_3 = \{d_1',d_2'\}$ constitute a forbidden structure as described in Theorem 1.

(2) This is obvious.

(3) Suppose w has a leaf b in u. Since we have $\ldots a_1 \ldots b_1 \ldots b_2 \ldots a_2 \ldots$(or its reverse) in any consistent permutation of the columns, it would be impossible to place b anywhere in the arrangement.

Theorem 4. Consider the case that M satisfies the *COP* and there is only one special terminal node t. Let w be a child of a node (other than m) on the path from m to the root of T such that w itself is not on the path. Then w must be empty.
Proof. Suppose w has a leaf b in u. Let a_1, a_2 be any leaf of the two endmost children of t, respectively and b', a leaf in a full child of t. Then we shall have $a_1 \ldots b' \ldots a_2$ (or its reverse) in any consistent permutation of the columns. But then, it would be impossible to place b anywhere in the arrangement.

By considering the terminal nodes we have a global view of the distribution of all full nodes by Theorems 3 and 4. Figure 3 illustrates an example with two terminal

nodes. Note that, in this example, we have flipped all full children of nodes on the terminal path down.

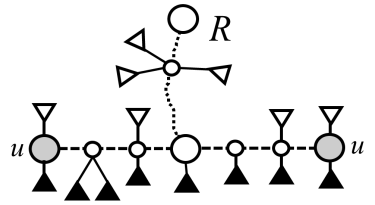


Fig. 3. The unique separating path between two terminal nodes *u* and *u'*

4. Constructing the New *PC*-Tree

Rather than using the node-by-node tree modification in the *B&L* algorithm, we shall update the current *PC*-tree to the new one in a batch fashion.

When a new row *u* is added, perform the following labeling operation:

1. label all leaf columns in *u full*.
2. the first time a node becomes partial or full, report this to its parent.
3. the first time a node *x* gets a partial or full child label *x partial*
4. the first time all children of a node *x* become full label *x full*

Our batch tree-update operation consists of the following steps:

1. delete all edges on the terminal path
2. duplicate the nodes on the terminal path (this is called the *splitting operation*)
3. create a new *C*-node *w* and connect *w* to all duplicated nodes according to their relative positions on the terminal path as follows (this is called the *modifying operation*): connect *w* directly to all *P*-nodes; connect *w* to all full children of *C*-nodes according to their original ordering; contract all degree two nodes

The following figures illustrate the splitting and the modifying operation on the graph of figure 3. Note that the root in a *PC*-tree does not play a major role in these operations.

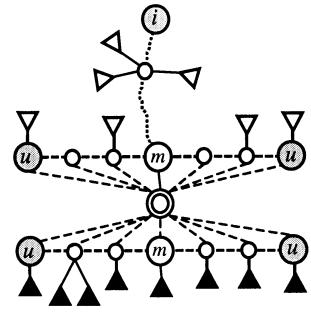


Fig. 4. The splitting operation

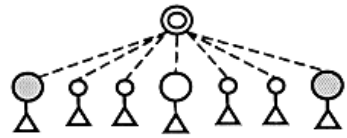


Fig. 5. Connecting to the new C-node

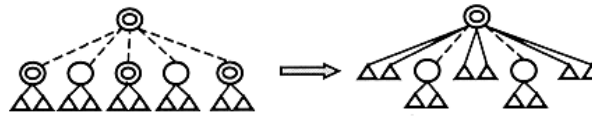
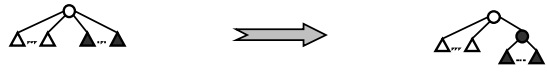


Fig. 6. The modifying operation

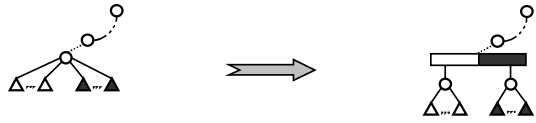
Theorem 5. The corresponding *PQ*-tree of the newly constructed *PC*-tree is the same as the one produced by the *B&L* algorithm.

Proof. As noted before, the *B&L* algorithm modifies the tree in a node-by-node bottom-up fashion based on 9 templates. Whenever a pattern is matched, it is substituted by the replacement.

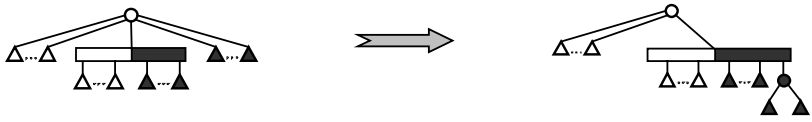
We shall prove the theorem by induction on the depth of the pertinent subtree. For each template operation of the *PQ*-tree, we shall illustrate the corresponding splitting operation of the *PC*-tree and show the equivalence of the two operations in terms of the resulting *PQ*-tree. We shall skip the easy cases of *P*0 and *P*1, *Q*0 and *Q*1.



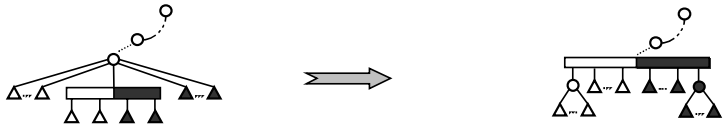
Template *P*2 for *ROOT* (*T,S*) when it is a *P*-node



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



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



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

Fig. 7. The template operations of Booth and Lurker's algorithm

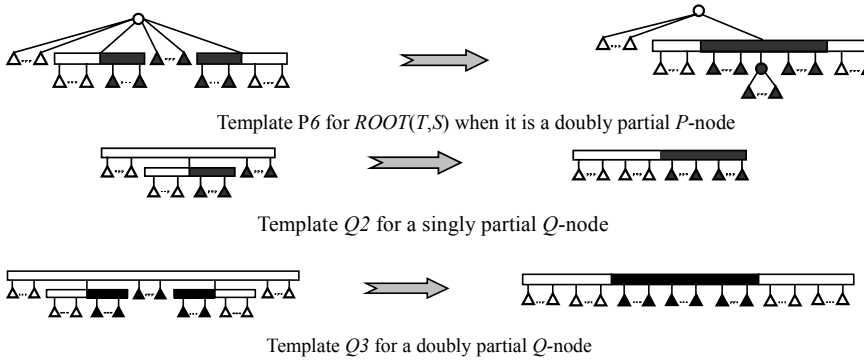


Fig. 7. (Continued)

First, consider templates at the root of *T*. In Figure 8 we consider template P2. The top part of Figure shows the *PQ*-tree replacement. The bottom part gives the corresponding *PC*-tree splitting and modifying operation. The equivalence is shown by the two rightmost diagrams, in which a *PQ*-tree (on top) and its corresponding *PC*-tree (in the bottom) are obtained through *B&L* template matching and the *PC*-tree operation, respectively.

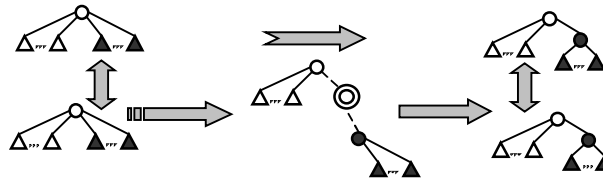


Fig. 8. Template *P*2 for *ROOT* (*T,S*) when it is a *P*-node when it is a *P*-node

The same goes for the templates P4, P6, Q2 and Q3 as shown in Figures 9,10,11,12.

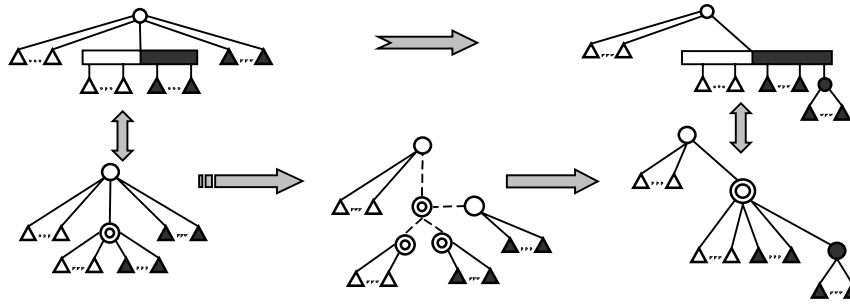


Fig. 9. Template *P*4 for *ROOT*(*T,S*) when it is a *P*-node with one partial child (*T,S*)

Now, assume the theorem is true for *PQ*-trees whose pertinent subtree is of depth k and consider a tree whose pertinent subtree is of depth k+1. In the *PQ*-tree operation, a template will be applied to a non-root terminal node x of the current *PQ*-tree T_1, to obtain a new *PQ*-tree T_2. To simplify the argument we assume the pertinent subtree of T_2 has depth k. Since, by induction, further *PQ*-tree operations on T_2 can be done equivalently through the corresponding *PC*-tree operations, we only have to show that the template matching operation on node x results in the same subtree as the one obtained by operation on the corresponding *PC*-tree of T_2. These are illustrated in Figures 13 and 14.

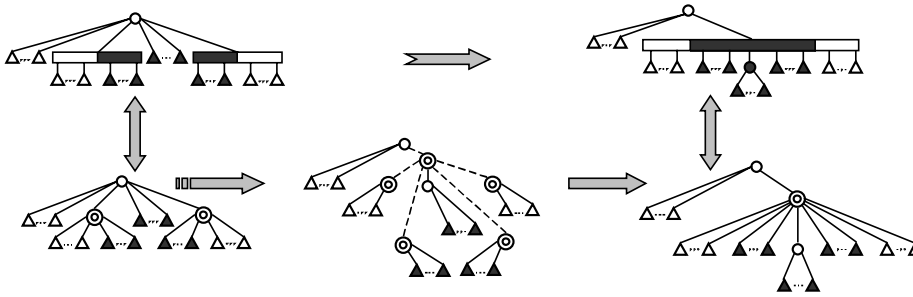


Fig. 10. Template P*6* for *ROOT*(*T*,*S*) when it is a doubly partial *P*-node

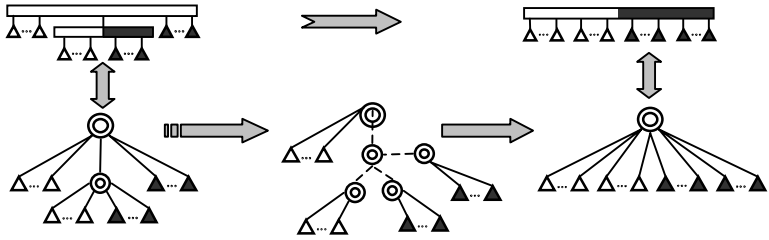


Fig. 11. Template *Q2* for a singly partial *Q*-node

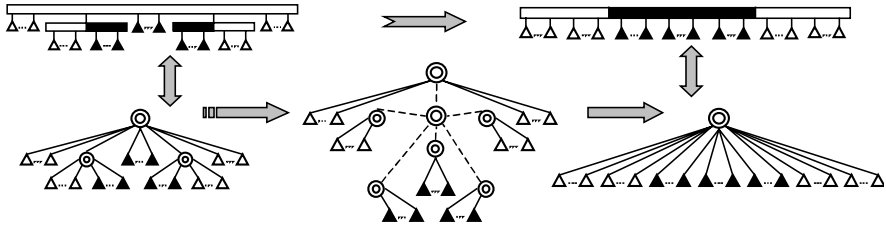


Fig. 12. Template *Q3* for a double partial *Q*-node

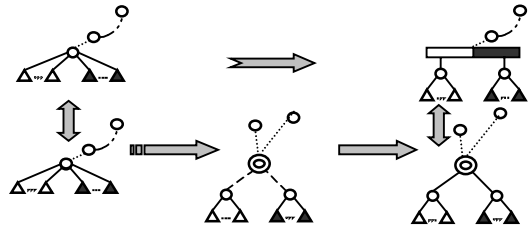


Fig. 13. Template *P*3 for a singly partial *P*-node which is not *ROOT* (*T*,*S*)

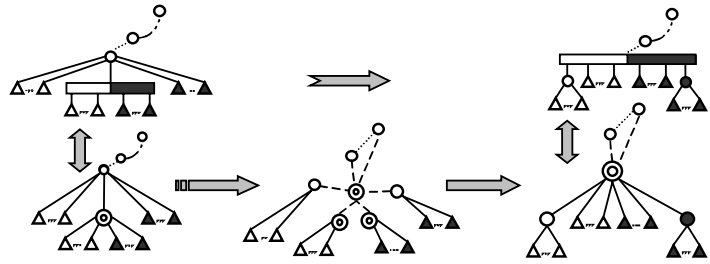


Fig. 14. Template *P*5 for a singly partial *P*-node, other than *ROOT*(*T*,*S*), with one partial child

End of Proof of Theorem 5

5. The Complexity Analysis

In the *B&L* algorithm, the bottom-up propagation potentially could examine the same node many times. Hence, special care must be taken to ensure efficiency. In our splitting and modifying operation of *PC*-trees the update is done in one batch.

We shall apply the same pointer strategy as in the *B&L* algorithm with regard to *P*-nodes and *C*-nodes. Our major saving is in skipping the pattern-matching phase at each node. The remaining replacement phase will be the same as in the B&L algorithm except that we use *C*-nodes instead of *Q*-nodes.

The detailed operations needed for *PC*-trees are all included in those of the *B&L* algorithm. Hence, the running time is linear in the size of the input.

6. Conclusion

We believe that a more efficient implementation of our *PC*-tree algorithm deviating completely from the pointer operation of B&L algorithm is possible. However, the purpose of this paper is to demonstrate that, conceptually, there is a simpler way to view the *PQ*-trees and their updates. Implementation issues of *PC*-trees will be left for those who have the experience of implementing the *B&L* algorithm.

7. Acknowledgement

This research was supported in part by the National Science Council under Grant NSC 89-2213-E-001.

References

1. K.S. Booth and G.S. Lueker, Testing of the Consecutive Ones Property, Interval graphs, and Graph Planarity Using PQ-Tree Algorithms, J. **Comptr. Syst. Sci**. 13, 3 (1976), 335-379.
2. D.R. Fulkerson and O.A. Gross, Incidence Matrices and Interval Graphs, **Pacific Journal of Math**., (1965), 15:835-855.
3. M. C. Golumbic, **Algorithmic Graph Theory and Perfect Graphs,** Academic Press, New York, 1980.
4. P.N. Klein and J.H. Reif, An efficient parallel algorithm for planarity, **J. of Computer and System Science** 37**,** (1988), 190-246.
5. N. Korte and R. H. Möhring, An Incremental Linear-Time Algorithm for Recognizing Interval Graphs, **SIAM J. Comput**. 18, 1989, 68-81.
6. A. Lempel, S. Even and I. Cederbaum, An Algorithm for Planarity Testing of Graphs, **Theory of Graphs**, ed., P. Rosenstiehl, Gordon and Breach, New York, (1967), 215-232.
7. W.K. Shih and W.L. Hsu, Note A new planarity test, **Theoretical Computer Science** 223, (1999), 179-191.