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 [7] where they represent partial embeddings of planar graphs. *PQ*-trees were invented by Booth and Lueker [1] 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 linear time *PC*-tree algorithm with the following advantages: (1) it does not use any template; (2) at each iteration, it does all necessary tree-modification operations in one batch and does not involve the node-by-node bottom-up matching; (3) it can be used naturally to test the circular ones property in matrices; (4) 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 linear time *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 elements 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. It should be noted that *S&H*'s approach is entirely different from that of using *PQ*-trees in *B&L*'s modification [1] to Lempel, Even and Cederbaum's (*LEC*) planarity test [6]. *B&L* used *PQ*-trees to test the consecutive ones property of all nodes adjacent to the incoming node in their vertex addition algorithm. In *LEC*, the leaves of the *PQ*-tree must be those nodes adjacent to the incoming node. Internal nodes of the *PQ*-tree are not the original nodes of the graph. Rather, they are there only to keep track of feasible permutations. Whereas in *S&H*, every *P*-node is an original node of the graph, every *C*-node represents a biconnected component in the partial embedding, and nodes adjacent to the incoming node can be scattered anywhere (both as internal nodes and as leaves) in the *PC*-tree. Thus, in *S&H*'s approach, a *PC*-tree faithfully represents a partial planar embedding of the given graph and is a more natural representation.

In this paper we shall focus on the application of *PC*-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 its columns can be permuted so that 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*. Figure 1 gives an example of a matrix satisfying the *COP* and its corresponding *PQ*-tree.

$$\begin{array}{c} 1\,2\,3\,4\,5\,6 \\ \left(\begin{array}{cccccc} 1&1&0&0&0&0 \\ 0&1&1&0&0&0 \\ 1&1&1&1&0&0 \\ 0&0&0&1&1&1 \\ 0&0&0&0&1&1 \end{array}\right) \end{array}$$

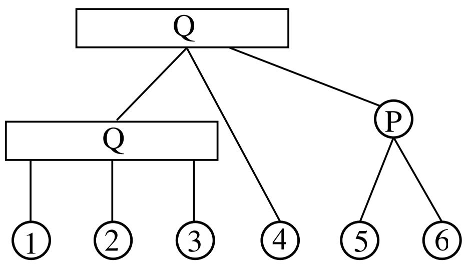


Figure 1. A *PQ*-tree and the consecutive ones property

1.1. A Brief Introduction of the original *PQ*-Tree Algorithm of *B&L*

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 (a new iteration starts), 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* ~ *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. In Section 3, we shall describe a more global strategy that does the tree-modification operation in one batch operation at each iteration.

A node is said to be *full* if all of its descendant leaves are in *u*; It is *empty* if none of its descendant leave s are in *u*; If some but not all of the descendant leave s 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] as shown in Figure 9. Two passes are used in the actual implementation. The first identifies the nodes to be processed and the second applies the templates.

A naïve implementation would classify each node as empty, full, or partial and then perform a pattern replacement. This will run into the danger of processing the entire tree even if the sets being reduced are small at each iteration. In order to achieve optimal efficiency, *B&L* 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 the pointers from their endmost siblings. In *B&L* algorithm, they adopted the notion of blocked, unblocked nodes and a block of blocked nodes in dealing with the borrowing of parent pointers. These measures unavoidably complicate the implementation of the algorithm. In Section 5, we shall adopt an alternative strategy of "parallel search", which identifies a parent pointer for each node immediately and costs at most $3|u|$ collectively at each iteration. Parallel search considerably simplifies the implementation.

1.2. Definition of *PC*-Trees

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. The frontier of a *PC*-tree *T*, denoted by *FRONTIER*(*T*), is the circular permutation of the columns obtained by ordering the leaves of *T* in a counter-clockwise circular order. Such a permutation is called a *consistent circular permutation*. Similar to the *PQ*-tree notations, two *PC*-trees are said to be *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、 Change the circular order of the neighbors of a *C*-node from clockwise to counter-clockwise.

Denote the set of all consistent circular permutations (or frontiers) of a *PC*-tree *T* by *CONSISTENT* (*T*) = $\{$*FRONTIER* (*T*') $|$ $T' \sim T\}$. Given a *PC*-tree *T* and a row *u* of *M*, define a new tree called the *u-reduction* of *T* to be one whose consistent circular permutations are exactly the original permutations in which the leaves in *u* occur consecutively in the circular list. This new tree is denoted by *REDUCE*(T,u).

$$\begin{array}{c} 1\,2\,3\,4\,5\,6 \\ \left(\begin{array}{cccccc} 1&1&0&0&0&0 \\ 0&1&1&0&0&0 \\ 1&1&1&1&0&0 \\ 0&0&0&1&1&1 \\ 1&0&0&0&1&1 \end{array}\right) \end{array}$$

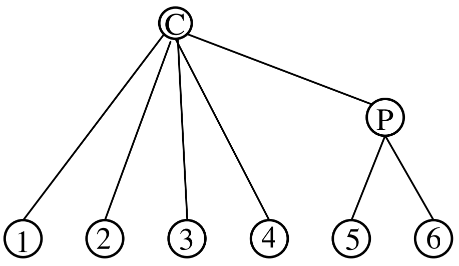


Figure 2. A *PC*-tree and the circular ones property

For each *PQ*-tree, we can obtain its corresponding *PC*-tree by replacing each *Q*-node w with a *C*-node and merging the parent x of w into the left-right ordering of the children of w (make x a neighbor of the two endmost children of w) to form a circular ordering. Note that flipping a *Q*-node in a *PQ*-tree is the same as changing the circular neighbor 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 the two endmost children of the root Q-node must remain endmost in any left-right ordering 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) Unlike a PQ-tree, the root of a *PC*-tree in general is not essential except to maintain the child-parent relationships.

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 template examination is replaced by *one* swift batch operation. Compared to the two-pass strategy of B&L, one might view our approach as a one-pass strategy that involves only node labeling operation. The second pass is replaced by a trivial tree-replacement operation.

In summary, our *PC*-tree algorithm has the following advantages: (1) it does not use any template; (2) at each iteration, it does all necessary tree-modification operations in one batch and does not involve the node-by-node bottom-up matching; (3) it can be used naturally to test the circular ones property in matrices; (4) the induced *PQ*-tree algorithm can considerably simplify *B&L*'s modification of Lempel, Even and Cederbaum's planarity test.

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 2.1. The following structure is forbidden in any consistent circular permutation of the columns of M that preserves the *CROP*. 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, a_3, b_1, b_2 and b_3 are distinct from each other.

(2) No two columns in any S_i are separated by columns in the other two subsets in any consistent

circular permutation. Namely, columns cannot be arranged in the following order, where j, k are distinct from i (but j could be equal to k):

(a column in S_i)…(a column in S_j)…(a column in S_i)…(a column in S_k)…

(3) Furthermore, b_1, b_2 and b_3 are not separated by any of the a_i's, namely, columns must be arranged in the following order, where i, j, k, (similarly, p, q, r) are all distinct from each other.

$a_i \ldots a_j \ldots a_k \ldots b_p \ldots b_q \ldots b_r \ldots$

Proof. From (2), we must have the following arrangement $a_1 \ldots b_1 \ldots b_2 \ldots a_2 \ldots$ (or the reverse) in any consistent circular permutation of the columns. By (2) and (3), one can only place a_3, b_3 into the above arrangement as follows:

I. $a_1 \ldots \boldsymbol{a_3} \ldots \boldsymbol{b_3} \ldots b_1 \ldots b_2 \ldots a_2 \ldots$

II. $a_1 \ldots b_1 \ldots b_2 \ldots \boldsymbol{b_3} \ldots \boldsymbol{a_3} \ldots a_2 \ldots$

However, in (I), columns in S_1 are separated by other subsets; and in (II), columns in S_2 are separated by other subsets. ■

3. Terminal Nodes and Paths

Note that the definition of empty, partial and full nodes are all relative to a fixed root in the tree T. In the following we shall define a notion that is independent of the selection of the root of T. Pick any node w of T as a root, one can form a tree rooted at w, denoted by $T(w)$. Define a node t of T to be a *terminal node* if there exists a node r in T such that, in $T(r)$, t is partial and none of t's children is partial. In other words, each child of a terminal node is either empty or full relative to a particular rooted tree. As long as there is a partial node, there must exists a descendent terminal node. Terminal nodes play a major role in simplifying our tree-update operation.

Theorem 3.1. If the given matrix M satisfies the *CROP*, then there exist 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. Let r_1, r_2, r_3 be the roots, respectively, that make them terminal nodes. We claim that none of the subtree T_{ti} with root t_i within $T(r_i)$ can contain the other two terminal nodes. It suffices to show that the subtree T_{t1} with root t_1 in $T(r_1)$ does not contain t_2.

Suppose, on the contrary, T_{t1} contains t_2. Let ST_{t2} be the subtree with root t_2 within the subtree T_{t1}. Since t_2 is either full or empty in T_{t1}, r_2 must be contained in ST_{t2} (for otherwise, the subtree at t_2 within the tree rooted at r_2 would still be empty or full). Without loss of generality, assume t_2 is full in T_{t1}.

Consider the parent w of t_2 in T_{t1}. If $w = t_1$, then we have a case that t_2 is full in $T(r_1)$ and t_1 is empty in $T(r_2)$ and t_1, t_2 are adjacent. Then, we can conclude: (1) if r_3 is contained within the subtree rooted at t_1 in $T(r_2)$, then t_1 would be the only terminal node; and (2) if r_3 is contained within the subtree rooted at t_2 in $T(r_1)$, then t_2 would be the only terminal node. In both cases, we cannot have a distinct third terminal node t_3.

Hence, we can assume $w \neq t_1$. We claim that w would become a partial child of t_2 in $T(r_2)$ as shown in Figure 3 (due to the reversal of parent-child relationship). This would be contradictory to the fact that t_2 is a terminal node in $T(r_2)$. Note that making the tree rooted at r_2 would only reverse the parent-child relationships for nodes along the path from r_2 to r_1 in $T(r_1)$. To prove this claim, let x ($\neq t_2$) be another child of w in T_{t1}. Since t_1 is a terminal node in T_{t1}, w must also be full. Hence, x is full in T_{t1}. Furthermore, x will remain full in $T(r_2)$. However, t_1 must have an empty child y in T_{t1}, which will remain empty in $T(r_2)$. Since both x and y are descendents of w in $T(r_2)$, w must be partial in $T(r_2)$.

Now, 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 2.1.

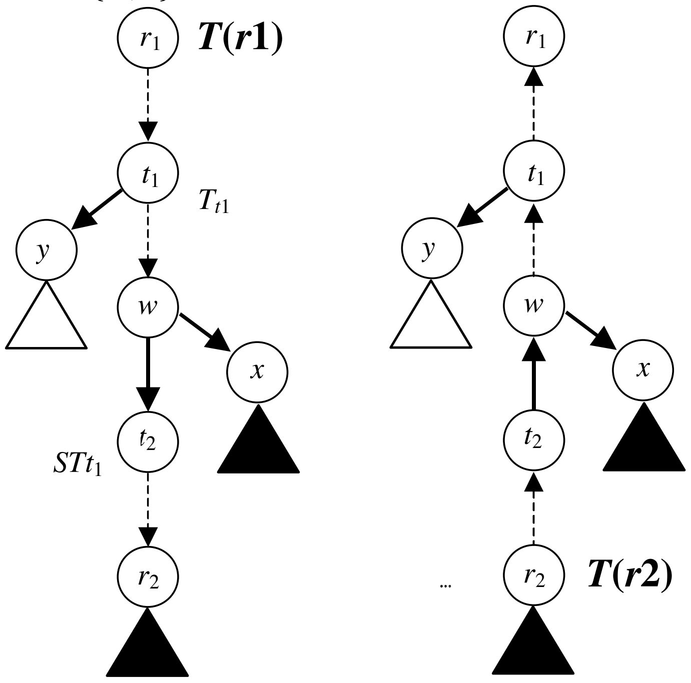


Figure 3. The tree rooted at r_1, the subtrees T_{t1} and ST_{t1}

∎

In case there is only one terminal node t, then in the tree rooted at t, every child of t is either full or empty. If t is a C-node, then the full children of t must be consecutive in its circular list and there is no tree modification necessary. If t is a P-node, then create a new P-node w as a child of t and change the parent of all full children of t to w.

The more interesting case occurs when there are two terminal nodes t_1 and t_2. Define the unique path in T connecting t_1 and t_2 to be the *terminal path*. 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 (change from clockwise to counter-clockwise) 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 3.2. If the matrix M satisfies the *CROP* and there are two terminal nodes t_1 and t_2 at some iteration u, then

(1) Every C-node in the interior of the terminal path satisfies that its full (respectively, empty) children together with its two neighbors on the path are contiguous in its circular list. Therefore, they can be flipped to one side correctly.

(2) If a terminal node, say t_2, is a C-node, then consider two cases: 1. t_2 is not on the unique path from the root to t_1. Then its full children together with its parent are contiguous in its circular list and exactly one of its parent's siblings is full; 2. t_2 is on that path. If t_2 is the root, then its full children together with its unique child, say w, on the path are contiguous in its

circular list and exactly one of w's siblings is full. If is not the root, then its full (respectively, empty) children together with its two neighbors on the path are contiguous in its circular list.

(3) Node m must be on the terminal path.

(4) Let $K = \{ x \mid x$ is a child of a node (other than m) on the unique path from m to the root of T such that x itself is not on this path $\}$. Then nodes in K are either all full or all empty.

Proof. Choose an empty child a_i and a full child b_i, respectively, from each of the t_i, $i = 1, 2$. 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. Let the two neighbors of w on the terminal path be s_1 and s_2. Suppose, in traversing the circular list of w from s_1 to s_2 through a full child d_2, we encounter an empty child d_1, 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. 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 2.1.

(2) This can be proved by arguments similar to those used in (1).

(3) Suppose otherwise. Let m' ($\neq m$) be the node on the path with maximum height. Then the subtree rooted at m' must contain the pertinent tree. However, m' is a descendant of m, a contradiction.

(4) First of all, we claim that K cannot contain any partial node since that would imply the existence of a third terminal node. Now, suppose there exist two nodes, x_1, x_2 in K such that x_1 is full and x_2 is empty. Since leaf nodes within the subtree rooted at m must be contiguous, they cannot be separated by x_1 and x_2. Hence, $S_2 = \{a_1,b_1\}$, $S_2 = \{a_2,b_2\}$ and $S_3 = \{x_1,x_2\}$ constitute a forbidden structure as described in Theorem 2.1. ■

By considering the terminal nodes and path, we have a global view of the distribution of all full nodes by Theorems 3.2. Figure 3 illustrates an example with two terminal nodes. In Figure 4, we have a case that all full children (whose corresponding subtrees are denoted by black triangles) of nodes on the terminal path are flipped away from the root. In Figure 5, all empty children are flipped away from the root. Exactly which is the case depends on case (4) of Theorem 3.2.

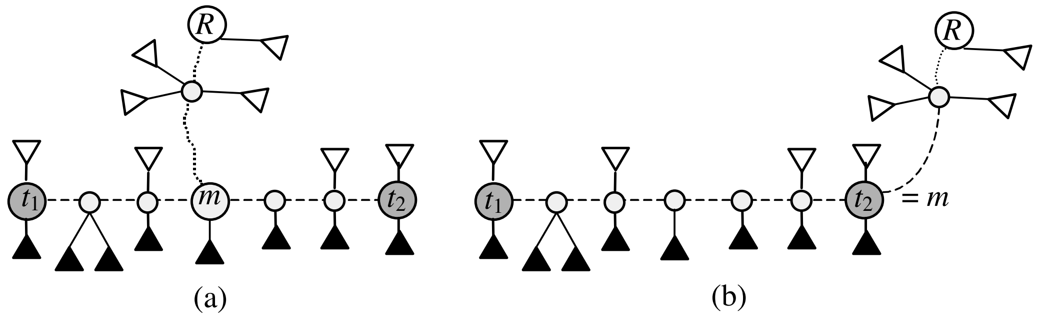


Figure 4. The unique terminal path between two terminal nodes t_1 and t_2

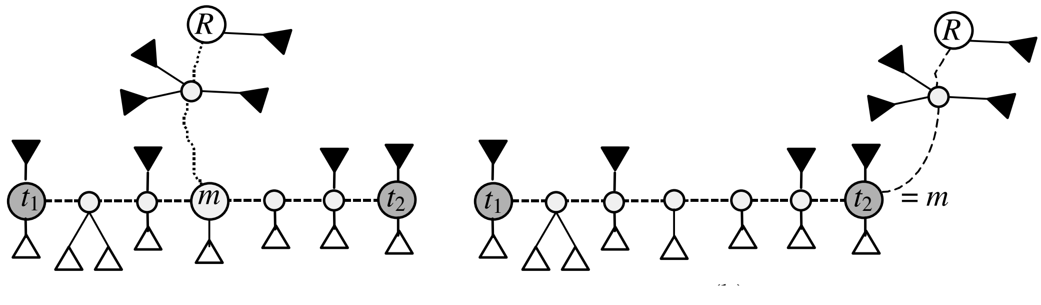


Figure 5. The terminal path with empty nodes flipped away from the root

3、 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. In the following we divide our algorithm into two stages: the labeling operation, and the tree-update operation. The latter consists of the splitting operation and the modifying operation.

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 tree-update batch operation consists of the following steps:

1. Delete all edges on the terminal path
2. The *splitting operation*: duplicate the nodes on the terminal path to create a cyclic order as shown in Figure 6. Contract all degree two nodes
3. The *modifying operation*: create a new *C*-node *w* and connect *w* to duplicated *P*-nodes and *C*-nodes according to their relative positions on the terminal path as shown in Figure 7. The detail is as follows: connect *w* directly to all *P*-nodes that have nonempty leaves; connect *w* to all full children of *C*-nodes according to their original ordering as shown in Figure 8.

In fact, by considering steps 2 & 3 together, we can see that every P-node on the path gets duplicated and every C-node on the path is deleted.

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

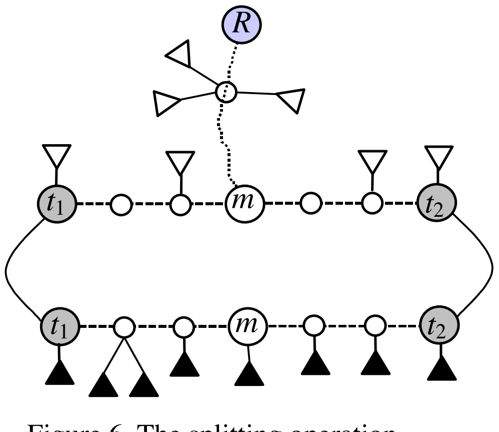


Figure 6. The splitting operation

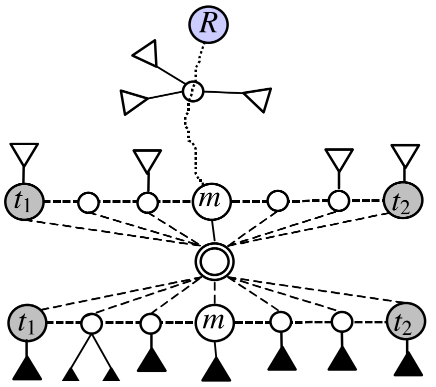


Figure 7. Connecting to the new C-node

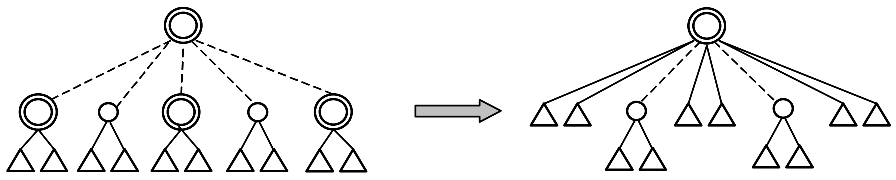
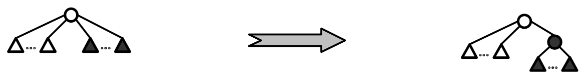


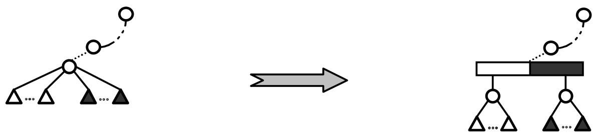
Figure 8. The modifying operation

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

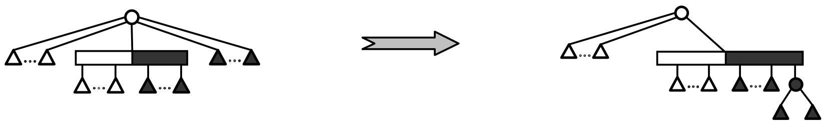
Proof. As noted before, *B&L* algorithm modifies the tree in a node-by-node bottom-up fashion based on the matching of 9 templates. Whenever a pattern is matched, it is substituted by the replacement. The templates used in *B&L* algorithm are illustrated in Figure 9. We shall skip the easy cases of *P*0 and *P*1, *Q*0 and *Q*1 [1].



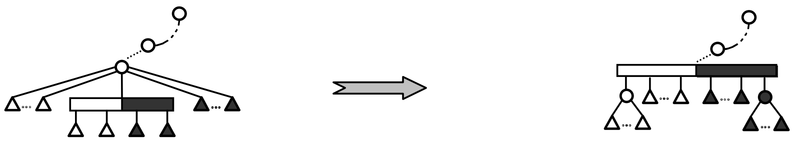
Template *P*2 for *ROOT* (*T,S*) when it is a *P*-node



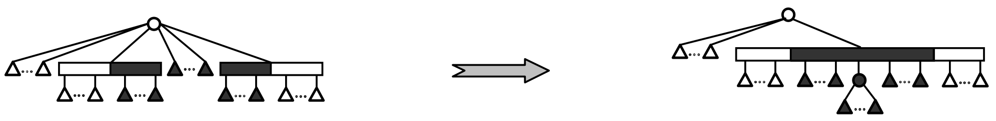
Template *P*3 for a singly partial *P*-node which is not *ROOT* (*T,S*)



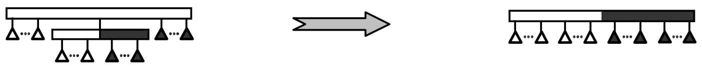
Template *P*4 for *ROOT*(*T,S*) when it is a *P*-node with one partial child (*T,S*)



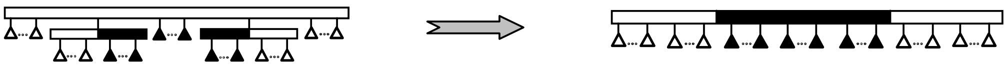
Template *P*5 for a singly partial *P*-node, other than *ROOT*(*T,S*), with one partial child



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



Template *Q2* for a singly partial *Q*-node



Template *Q3* for a double partial *Q*-node

Figure 9. The template operations of *B&L* algorithm

We shall prove the theorem by induction on the depth of the pertinent subtree at which *B&L* executed the template operation. 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. The basis of the induction at depth 1 occurs when the operation is performed on the children of *ROOT*(*T*,*u*).

In Figures 10 to 14 below we use the following convention: The two leftmost diagrams give a *PQ*-tree (on the top) and its corresponding *PC*-trees (at the bottom). The top part of each 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.

We first consider templates applied at *ROOT*(*T*,*u*). In Figure 10 we consider template P2.

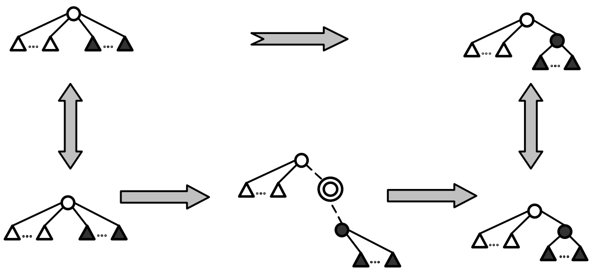


Figure 10. Template *P*2 for *ROOT* (*T*,*S*) when it is a *P*-node

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

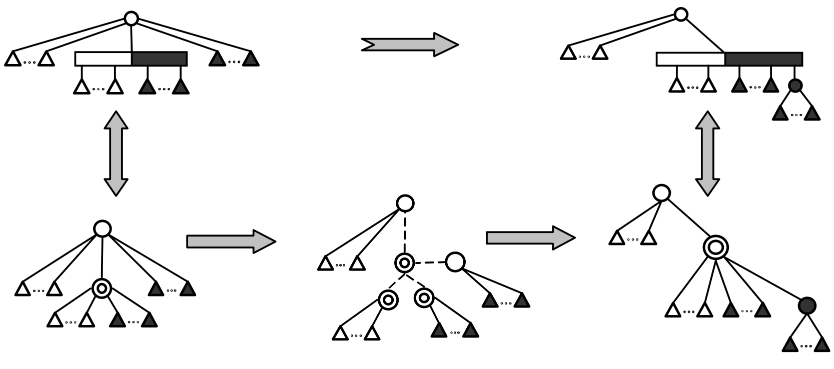


Figure 11. Template *P*4 for *ROOT*(*T*,*S*) when it is a *P*-node with one partial child (*T*,*S*)

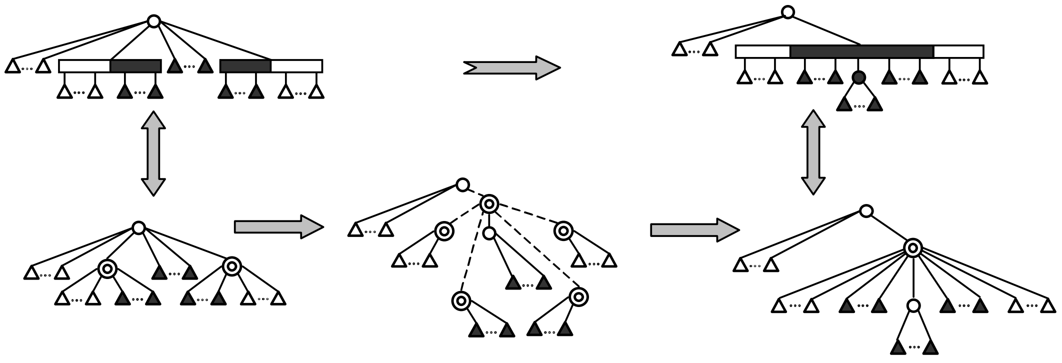


Figure 12. Template P*6* for *ROOT*(*T*,*S*) when it is a doubly partial *P*-node

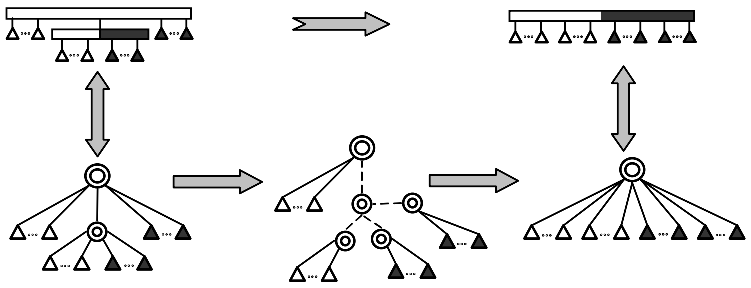


Figure 13. Template *Q2* for a singly partial *Q*-node

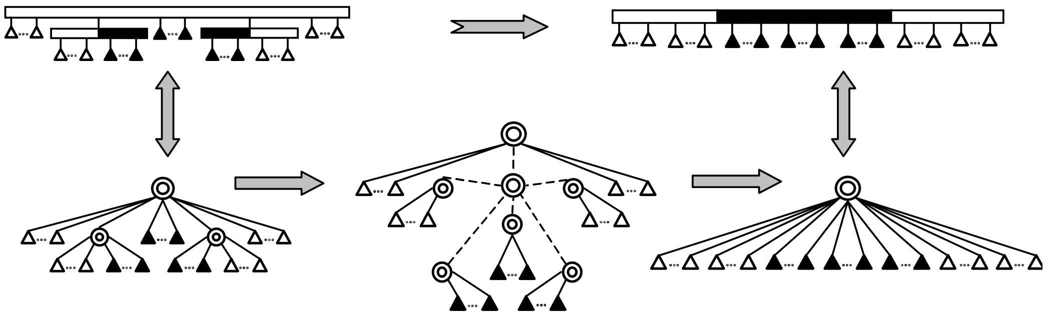


Figure 14. Template *Q3* for a double partial *Q*-node

The above has established the basis for the induction. 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 *B&L 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 further *PQ*-tree operations on T_2 can be done equivalently through the corresponding *PC*-tree operations by induction, we only have to show that the PC-tree operation performed on T_1 results in the same *PC*-tree performed on T_2. In Figures 15 and 16 below we describe the *PC*-tree operations: (a) Transform the current *PQ*-tree into a *PC*-tree; (b) Illustrate the *PC*-tree operation on the entire tree. However, the part that is related to this particular node is highlighted with the remaining part (which is the same for all diagrams) indicated by thick dotted line. Note that, with the bottom-up strategy of *B&L* algorithm, the node in consideration must be a terminal node in the *PC*-tree, which is at the very end of the splitting operation. (c) The final modifying operation where the newly created *C*-node, say *w*, is connected directly to all *P*-nodes that have nonempty leaves and *w* is connected to all full children of *C*-nodes according to their original ordering (the original *C*-nodes on the terminal path are deleted). The original *PQ*-tree templates are shown by the leftmost two diagrams.

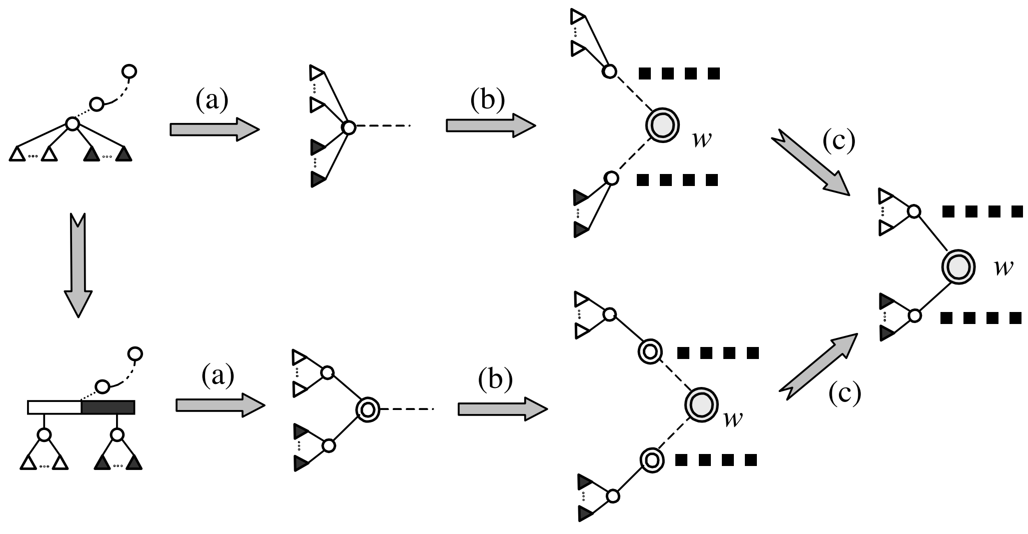


Figure 15. Template *P*3 for a singly partial *P*-node which is not *ROOT* (*T,S*)

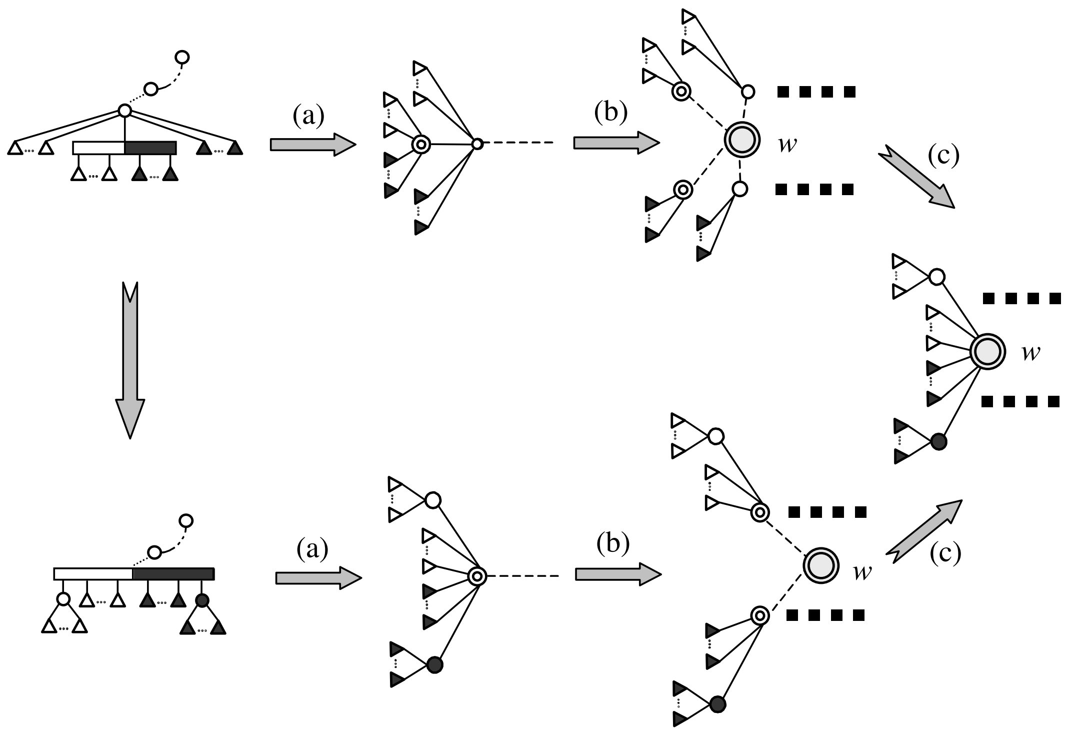


Figure 16. Template *P*5 for a singly partial *P*-node, other than *ROOT*(*T*,*S*), with one partial child

End of Proof of Theorem 4.1. ■

Corollary 4.2. The *PC*-tree algorithm can be used to test the circular ones property.

Proof. Theorem 4.1 compared the *PC*-tree algorithm versus the *PQ*-tree algorithm concerning the consecutive ones testing and showed that they result in the same *PQ*-tree at each iteration. For the *CROP* test, the difference is that, at some iteration, we might encounter a row *u* whose complementary column set satisfies the *COP*. By Theorem 3.1, one can still obtain the terminal nodes and the terminal path for such a row. However, at this time, the empty nodes (rather than the full nodes) are flipped away from the root as illustrated in Figure 4. Since our splitting and modification operations are completely independent of the root of the tree, such a change does not affect our *PC*-tree modification.

On the other hand, *B&L* algorithm uses templates based on the consideration of pertinent nodes. Therefore, they could not afford to consider templates based on the consideration of empty nodes (which is exactly what is needed in this case if one adopts the template matching strategy). Hence, their implementation cannot be used to test the *CROP* directly and a transformation on the matrix itself is needed to achieve that [3]. ■

5. Implementation Details

In the last paragraph of Section 1.1, we mentioned an important implementation trick in *B&L*'s *PQ*-tree algorithm -- the borrowing of parent pointers. We shall do the same in our *PC*-tree implementation. Namely, parent pointers are only kept for children of *P*-nodes and for endmost children of *C*-nodes; parent pointers for internal children must be borrowed from the endmost ones through their siblings. However, in identifying the parent for children of a *C*-node, we shall adopt the following "parallel search" strategy, which considerably simplifies the implementation.

The notion of parallel search is defined within a circular linked list of a *C*-node. Let C_1 be a *C*-node and *t* be a node (which does not have a parent pointer) in the circular linked list of C_1. A *parallel search* (*PS*) from *t* is a traversal in the circular linked list from *t* along both the clockwise and the counter-clockwise directions in equal pace. Every node traversed will be marked. The purpose of the *PS* is to identify the parent for *t*, namely C_1. Let v_1 and v_2 be the two endmost children of C_1 that have parent pointers to C_1. The very first *PS* in C_1 from a node will terminate when one of the endmost children is reached. Afterwards, a *PS* will terminate when an endmost child is reached or a previously searched node is reached. In both cases, the parent for *t* can be identified. Since the time spent in each iteration should be proportional to the number of incoming columns, $|u|$, we shall terminate a *PS* after $|u|$ steps.

There are three cases that could occur at the end of the labeling operation for the node C_1: (1) all of its children are labeled; (2) a contiguous subset of its children including one of its endmost children are labeled; (3) a contiguous subset of its children are labeled but none of its endmost children is labeled. In both cases (1) and (2), the total cost of all *PS*s occurred for the node C_1 is proportional to the number of labeled children of C_1. Case (3) can occur only when C_1 is *ROOT*(*T*,*u*), which incur a cost of $/u|$ and can happen at most once in the iteration for *u*. Therefore, if the cost for parallel search is charged separately, we could assume each child of C_1 can get a parent pointer in constant time. This is exactly what we have assumed in the labeling operation in Section 4.

Now, consider the tree-update operation. *C*-nodes on the terminal path are deleted and *P*-nodes are duplicated. Each node on the path has at least one child that contains a distinct leaf column in $|u|$. Hence, the number of duplicated nodes (or the number of nodes on the terminal path plus one - the newly created *C*-node) is no greater than $|u| + 1$. The sibling relationships of the new *C*-node can be constructed by connecting *P*-nodes and the neighboring children of *C*-nodes on the path according to their order on the path (as shown in Figure 17), which takes time proportional to the number of nodes on the path (and bounded by $|u|$).

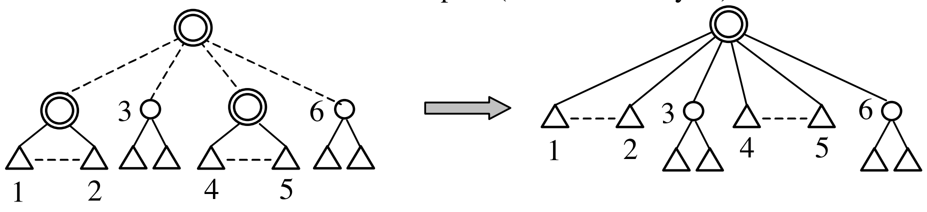


Figure 17. The construction of the new sibling relationships

6. Complexity Analysis

Since the number of new nodes generated is $O(|u|)$, the number of steps involved in both the labeling operation and the tree-update operation are bounded by $O(|u|)$, the running time at iteration u is bounded by $O(|u|)$. Hence, the time complexity of the algorithm is linear.

Compared to the *B&L* algorithm, our major saving is two folds: first, we have avoided the pattern-matching phase at each node; secondly, our parallel search strategy has considerably simplified the borrowing of parent pointers for internal children of C-nodes.

7. Acknowledgement

We are grateful for the generous support from the National Science Council under Grant NSC 90-2213-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.