# Parallel Recognition of Complement Reducible Graphs and Cotree Construction†

*D.G. Kirkpatrick,*
*T. Przytycka*

Technical Report 88-1
January 1988

**Abstract**

A simple parallel algorithm is presented for constructing parse tree representations of graphs in a rich family known as cographs. From the parse tree representation of a cograph it is possible to compute in an efficient way many properties which are difficult for general graphs. The presented algorithm runs in $O(\log^2 n)$ parallel time using $O(n^3/\log^2 n)$ processors on a CREW PRAM.

† This research was supported in part by the Natural Sciences and Engineering Research Council of Canada.

## 1. Introduction

Recent development of parallel computation on trees ([MR85], [CV86b], [GR86], [ADKP87],[He86a],[B74]) has led to efficient parallel algorithms for a number of problems in some restricted classes of graphs. These classes include graphs which can be defined by certain composition rules. These composition rules make it possible to represent a graph from the given class in the form of a parse tree. Having a tree representation of such a graph one can use a tree contraction schema to compute efficiently some graph properties which are very difficult for general graphs. Such parallel algorithms have been presented for series parallel graphs (in [H86a]) and for cographs (in [ADKP87]). These algorithms assume that the graph is given in the form of a parse tree. This motivates the problem of designing efficient parallel algorithms to recognize membership in the given class and to construct the corresponding parse tree. He [H86b] solved this problem for the class of two terminal series parallel graphs (TTSP graphs). His algorithm constructs a binary decomposition tree if a given graph is a TTSP graph. Given a graph with $n$ vertices and $m$ edges the algorithm runs on a CRCW PRAM in $O(\log^2 n+\log m)$ parallel time using $O(n+m)$ processors.

In this paper we present an algorithm for parallel construction of a parse tree for complement reducible graphs (cographs). In fact the algorithm produces a special kind of parse tree called a cotree which is a unique representation of a cograph. The algorithm can also be used to determine whether or not a given graph is a cograph. The idea is similar to that of tree contraction except that the underlying parse tree is not assumed to be known in advance. The algorithm runs in $O(\log^2 n)$ parallel time using $O(n^3/\log^2 n)$ processors on a CREW PRAM. In the next three sections we outline an implementation using $O(n^3/\log n)$ processors. Section 6 describes a reduction to $O(n^3/\log^2 n)$ processors.

## 2. Definitions and notation

A *complement reducible graph,* also called a *cograph,* is defined recursively in the following way:

*(i)* A graph on a single vertex is a cograph;

*(ii)* If $G_1$, $G_2$ are cographs, then so is their union ; and

*(iii)* If G is a cograph, then so is its complement.

Cographs are easily seen to satisfy the following property (cf. [CLS81]) :

**Property 1.** An induced subgraph of a cograph is a cograph.

The class of cographs is a very rich class of graphs. Cographs arise in many disparate areas of mathematics (see [CLS81] for references). Cographs form precisely the class of graphs which do not contain $P_4$ as an induced subgraph ( $P_4$ is a path of four vertices). This characterization suggests a simple parallel algorithm for the *recognition* of cographs that operates in O(1) time using $O(n^4)$ CREW processors. Such an algorithm, however, is not guaranteed to reveal the simple recursive structure that is imposed by the cograph definition and exploited in many cograph algorithms.

The definition of a cograph suggests a natural parse tree representation. However this way of presenting a cograph may not be unique. A unique representation is provided by the so-called *cotree* [CLS81]. A cotree, $T_G$ , is the tree presenting the parsing structure of a cograph G in the following way :

- The leaves of $T_G$ are the vertices of G.

- The internal nodes of $T_G$ represent the operation complement-union (that is the graph associated with an internal node is the complement of the union of the graphs associated with its descendent nodes).
- Each internal node except possibly the root has two or more children. The root has only one child iff the graph is disconnected.

In order to simplify the description of algorithms which use the cotree representation of a cograph, each node $x$ of a cotree T is assigned a label, label($x$), in the following way:

- label(root) = 1; and
- if $y$ is a child of $x$ then label($y$) = 1 - label($x$).

Figure 1 illustrates a cograph G and its labeled cotree $T_G$. The labeling of a node $x$ records the parity of the number of complement-union operation on the path between $x$ and the root. It is easy to confirm that :

**Property 2.** Two vertices $u$ and $v$ in a cograph G are adjacent iff in the cotree defining G the lowest common ancestor of $u$ and $v$ is labeled 1.

To minimize confusion, we talk about *vertices* when we refer to a graph and about *nodes* when we refer to a tree.

The nodes of a cotree labelled by 0 are called *0-nodes* and those labelled by 1 are called *1-nodes*. We also use the following notation: $n$ denotes the number of vertices in G, $\Gamma_G(v)$ denotes the set of neighbours of the vertex $v$ in G, and $lca_T(v_1,v_2)$ denotes the lowest common ancestor of nodes $v_1$ and $v_2$ in the tree T (the subscripts G and T are omitted if it is obvious to which graph or tree we refer).

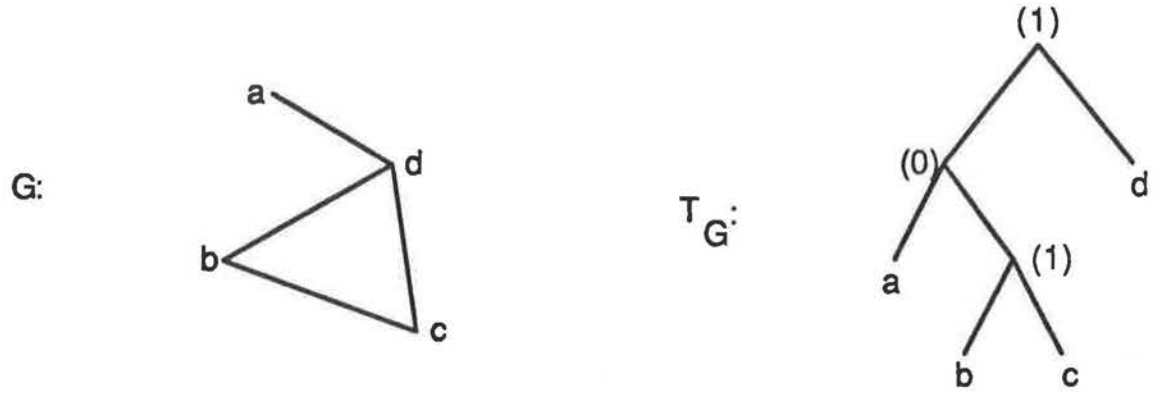



Figure 1 : A cograph and its cotree

Define the following relations between vertices of a graph G:

(1) $S_0(u,v) \Leftrightarrow \Gamma(v) - \{u\} = \Gamma(u) - \{v\}$ and $u,v$ are not adjacent,

(2) $S_1(u,v) \Leftrightarrow \Gamma(v) - \{u\} = \Gamma(u) - \{v\}$ and $u,v$ are adjacent,

(3) $Z_0(v,u,w) \Leftrightarrow$ (i) $\Gamma(v) \oplus \Gamma(w) = \{u\}$ , where $\oplus$ denotes the symmetric difference,

(ii) $\Gamma(u) \neq \Gamma(v)$, and

(iii) $w$ is not adjacent either to $v$ or to $u$,

(4) $Z_1(v,u,w) \Leftrightarrow$ (i) $\Gamma(v) \oplus \Gamma(w) = \{u\}$,

(ii) $\Gamma(v) \neq \Gamma(u)$, and

(iii) $w$ is adjacent to both $v$ and $u$.

The vertices satisfying relation $S_i$ (i = 0,1) are called *siblings*. The vertices satisfying $S_0$ are called *weak siblings* and the vertices satisfying $S_1$ are called *strong siblings*.

**Lemma 1.** Two leaf nodes $v_1$ and $v_2$ have the same father in the cotree $T_G$ iff the corresponding vertices $v_1$ and $v_2$ are siblings in G.

**Proof :** (=>) Assume that $v_1$ and $v_2$ have the same father in the cotree T. Note that for any vertex $v$ such that $v \neq v_1$ and $v \neq v_2$ , $\text{lca}(v,v_1) = \text{lca}(v,v_2)$. Hence, by property 2, any vertex adjacent to $v_1$ is adjacent to $v_2$ and $\Gamma(v_1) - \{v_2\} = \Gamma(v_2) - \{v_1\}$.
(<= ) Assume that $v_1$ , $v_2$ have different fathers. Let $v = \text{lca}(v_1,v_2)$. At least one of the paths from $v_i$ to v ($i$ = 1,2) in cotree T is longer than one. Assume w.l.o.g. that the path from $v_1$ to $v$ is longer then one. We can find on this path an internal node $u$ which is labelled differently than $v$. So, there exists a node $w$ ($w \neq v_1$ and $w \neq v_2$) such that $\text{lca}(w,v_1) = u$ and $\text{lca}(w,v_2) = v$. But this means that $w$ is connected to exactly one of $v_1$, $v_2$ . So neither $S_0(v_1,v_2)$ nor $S_1(v_1,v_2)$ holds.

A maximal set of weak siblings is called a *0-bunch set* and a maximal set of strong siblings is called a *1-bunch set.* A smallest connected subgraph of the cotree T containing a 0-bunch set is called a *0-bunch* and a smallest connected subgraph of the cotree containing a 1-bunch set is called a *1-bunch* (see Figure 2). The vertex with the smallest index among the vertices in a bunch set is called the *representative* of this set.


0-bunch 1-bunch

Figure 2

If we replace a bunch set in a cograph G by its representative, say $v$, then, by property 1, the graph G' = (V',E') obtained in such a way is also a cograph. Consider the following construction of a tree T' from the tree T:

(1) If vertices in the bunch set are the only children of some internal node then substitute the representative of the bunch set for the whole bunch to which they belong (see figure 3a)).

(2) If the vertices in the bunch set are not the only children of some internal node then remove from T all vertices in this set but the representative (see figure 3b)).

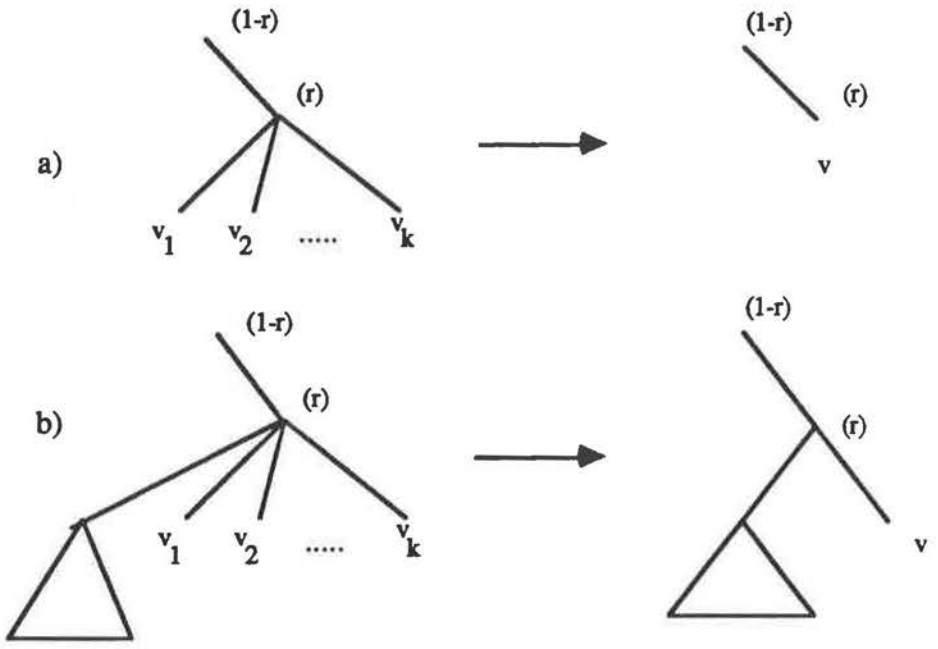



Figure 3

**Note:** The difference in the labels of the parent of the representative of a bunch and the bunch type in these two cases ensures that this substitution is reversible.

**Lemma 2.** The tree T' obtained from the tree T in the way described above is the cotree of the cograph G'.

**Proof:** Note that for $u,w \in V'-\{v\}$ $label(lca_T(u,w)) = label(lca_{T'}(u,w))$. Also $label(lca_T(u,v)) = label(lca_{T'}(u,v))$. So, by the definition of G' and property 2, the tree T' is the cotree of G'.

Similar to siblings , the vertices in relation $Z_i$ (i = 0,1) have a special position in the cotree. It is specified by the following lemma :

**Lemma 3:** The relation $Z_0(v,u,w)$ holds iff $v,u$, and $w$ are positioned in the cotree as illustrated in figure 4a. Similarly the relation $Z_1(v,u,w)$ holds iff $v,u$, and $w$ are positioned in the cotree as illustrated in figure 4b.

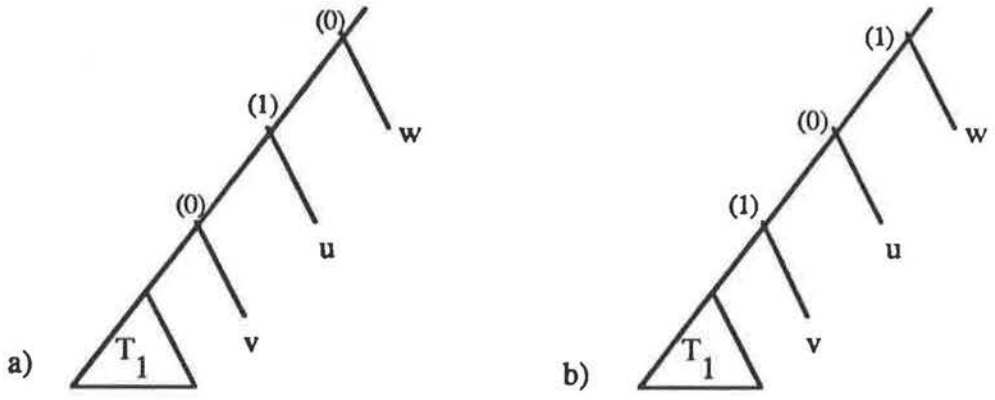



Figure 4

**Proof :** We will prove the lemma for the relation $Z_0$ only. The proof for the relation $Z_1$ is similar.

(=>) Assume that $v,u$, and $w$ are positioned as illustrated in figure 4a. By property 2, $v$ and $w$ have no neighbours in $T_1$. Assume $t \notin T_1$ and $t \neq v,u,w$ then $lca(v,t) = lca(w,t)$ and (i) follows. As immediate consequences of the definition of cotree we have (ii) and (iii).

(<=) From (i) and (iii) we have that $v$ and $w$ are not adjacent, $v$ and $u$ are adjacent, $u$ and $w$ are not adjacent. This implies the position of the nodes as in figure 5a. From (i) we know that there are no nodes between $a$ and $b$, $u$ and $a$ , $w$ and $b$, and from (ii) we have additionally $\Gamma(v) \neq \Gamma(u)$. This implies the more restricted position of the nodes shown in figure 5b. Finally point (i) restricts us to the position presented on figure 4a.

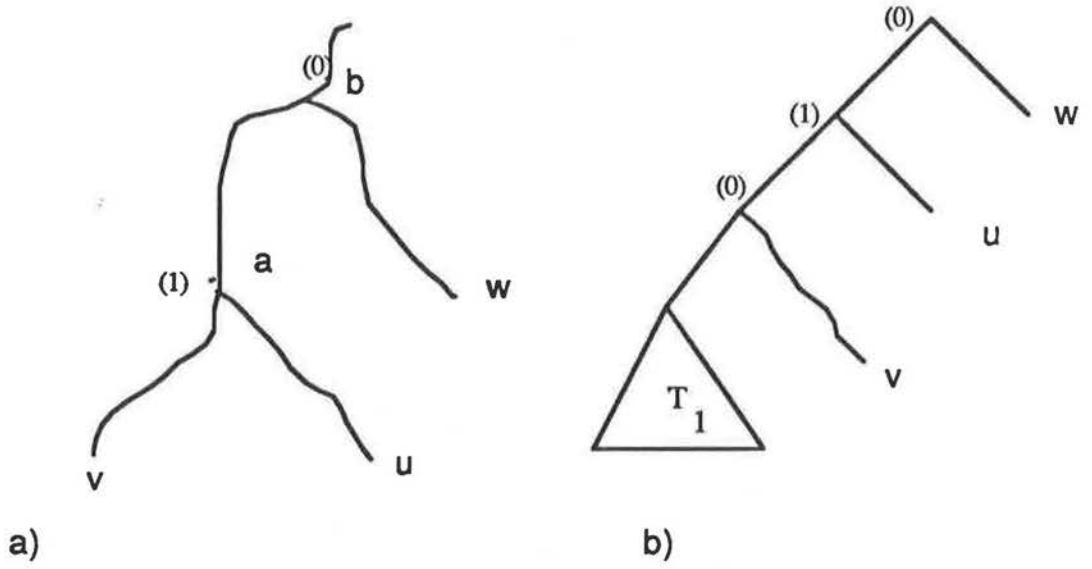



Figure 5

A vertex $u$ for which there exist vertices $v,w$ such that $Z_0(v,u,w)$ or $Z_1(v,u,w)$ is called *a contractible vertex* . The corresponding leaf in a cotree is a *contractible leaf* . If a node has a contractible leaf as a child then it has exactly two children one of them being a leaf and the other being a nonleaf.

A *contractible sequence* is a maximal sequence of distinct vertices (leaves in the cotree) $u_1,u_2,...,u_k$ such that there exist two vertices $v,w$ for which $Z_i(v,u_1,u_2)$, $Z_{1-i}(u_1,u_2,u_3)$ ,..., $Z_i(u_{k-1},u_k,w)$ all hold, and there does not exist an $x$ such that $Z_{1-i}(x,v,u_1)$ holds.

Define a *branching node* as an internal node having more than two children or having more than one leaf as a child. Note that any nonbranching node appears in the cotree as vertex $v$ in Figure 6.

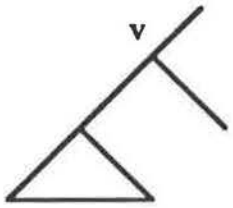
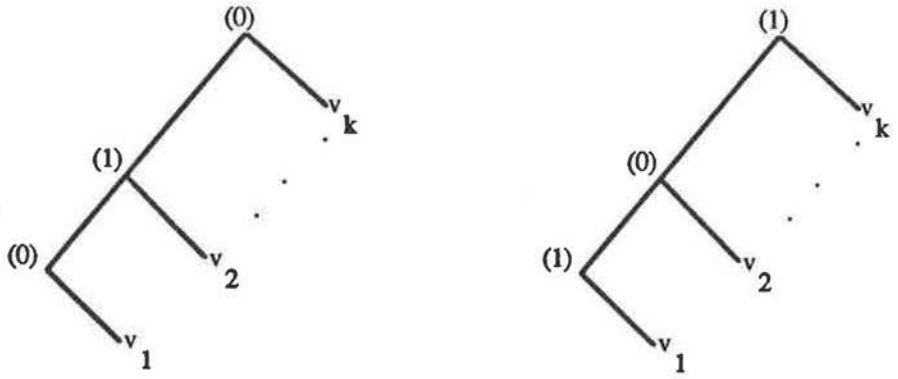



Figure 6

A smallest connected induced subgraph of a cotree containing a contractible sequence is called a *line* . A line is a *0-line* if the lowest level internal node is a 0-node and a *1-line* otherwise. By lemma 3, lines have the form presented in Figure 7. The set of vertices associated with a 0-line is called a *0-line set* and the set of vertices associated with a 1-line is called a *1-line set*. The leaf which has a smallest level in the cotree among other vertices in a line (i.e. vertex $v_1$ in Figure 7) is called the *representative* of the given line set.




a) 0-line

b) 1-line

Figure 7

Let G' be the cograph obtained from G by replacing a line set by its representative. Let T' be the tree obtained by removing from T all elements of a line set and their parents except the representative and its parent. The parent of the representative takes as its new parent the (former) parent of the highest level vertex in the line set (see Figure 8).

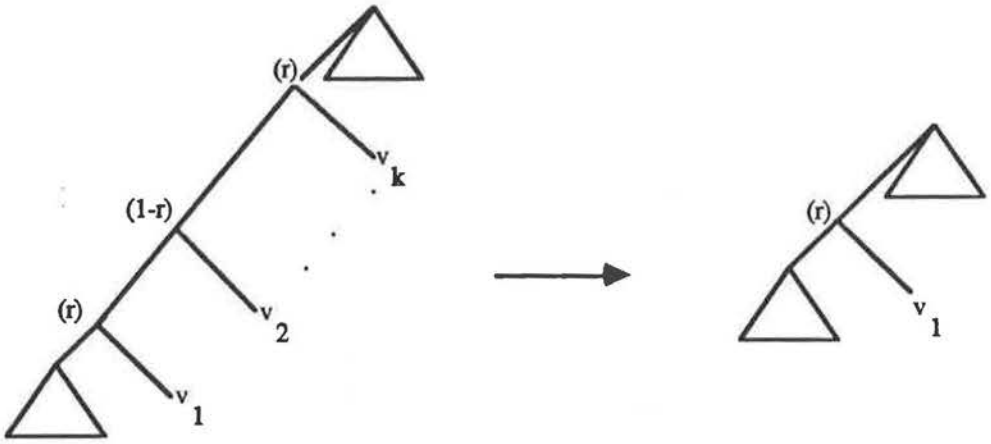



Figure 8

**Lemma 4.** The tree T' obtained from the tree T in the way described above is the cotree of the cograph G'.

**Proof :** Similar to the proof of the lemma 2.

In figure 9, $a$,$f$ and $g$ are branching nodes, $\{v_1,v_2,a\}$ and $\{v_8,v_9,g\}$ induce 0–bunches and $\{v_3,v_4,v_5,b,c,d\}$ induces a 1-line. The vertex $v_1$ is the representative for the first bunch and the vertex $v_8$ for the second. The vertex $v_3$ is the representative for the line.

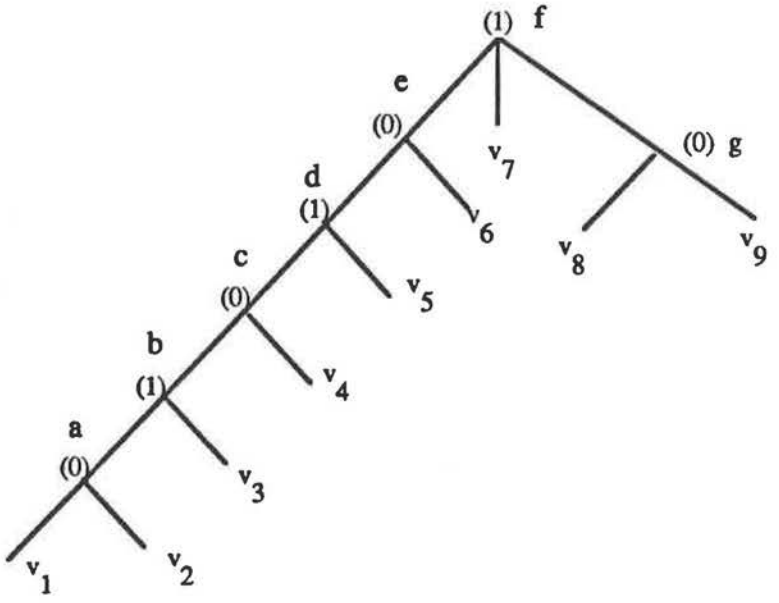



Figure 9

Note that in the above example all line sets and bunch sets are disjoint. This is true in general.

**Lemma 5.** Let each of U,W be a line set or a bunch set. If $U \neq W$ then $U \cap W = \varnothing$.

**Proof :** By lemmas 1 and 3, an element of a bunch set cannot belong to a line set. Note also that $S_i(w,u)$ and $S_j(u,v)$ implies $i = j$ and $S_i(v,u)$, so an element of a 0-bunch set cannot belong to a 1-bunch set. If U and W are both line sets or both bunch sets then $U \cap W \neq \varnothing$ contradicts the maximality of U and W.

## 3. The main idea

We will assume that the input graph is connected. If it is not we can run a parallel algorithm for finding connected components ([HCS79]) and join cotrees obtained for each connected component according to the cotree definition.

Let the input graph be $G_0 = (V_0,E_0)$. Assume for now that $G_0$ is a cograph and denote its cotree by $T_0$. The idea is to partition the set of vertices into subsets, remove from each subset all but one vertex (its representative) and reduce the problem to constructing the cotree for the graph induced on the diminished vertex set. Iterating this step we obtain a sequence of graphs $G_0 = (V_0,E_0)$, $G_1 = (V_1,E_1)$ ,...., $G_k = (V_k,E_k)$ such that $V_i$ is obtained from $V_{i-1}$ by performing a partition of $V_i$ and then removing all but one vertex in each set of the partition. We want the constructing sequence to have the property that having the cotree for $G_i$ one can easily construct the cotree for $G_{i-1}$. It seems natural at first to partition $V_i$ into bunch sets. Unfortunately the length of the sequence of graphs which is constructed in this way is proportional to the length of the longest path in the cotree $T_0$ which may be proportional to $|V_0|$ if $T_0$ is unbalanced. This is the reason for introducing line sets in addition to bunch sets into the partition.

By lemma 5, the set of vertices of a cograph can be partitioned into 0-bunch sets, 1-bunch sets, 0-line sets, 1-line sets and single vertex sets. For any set $U^i$ from such a partition, we can consider the smallest connected subtree of the cotree T which contains elements of this set as leaves. Such a subtree will be called a *fragment of T induced by this set*. Notice that in the proposed partition the only possible fragments are bunches, lines or single vertices.

The algorithm proceeds in stages. In stage i, it produces a triple $(G_i,U_i,F_i)$ such that the sequence of triples produced by the algorithm satisfy the following conditions :

(i) The first element of the sequence is the triple $(G_0,\{\{v\},v\in V_0\},\{V_0\})$,

(ii) $U_{i+1}=\{U_{i+1}^1,....,U_{i+1}^t\}$ is a partition of $V_i$,

(iii) $V_{i+1} = \{u_i^1,...,u_i^t\}$ where $u_i^j$ is the representative of $U_i^j$,

(vi) $G_i=(V_i,E_i)$ is the subgraph induced by $V_i$,

(v) $F_i = \{F_i^1,..., F_i^t\}$ where $F_i^j$ is the fragment of $T_{i-1}$ induced by $U_i^j$,

(vi) The last element in the sequence is the first triple $(G_k,U_k,F_k)$ for which $|U_k| = 1$.

Note that the cotree $T_k$ is just the only fragment in $F_k$. For i = k-1,...,1, we construct cotree $T_i$ from cotree $T_{i+1}$.

We define the operation *reduce* which i) partitions vertex set into bunch sets, line sets and single vertex sets, ii) finds representatives for those sets, iii) constructs corresponding fragments, and iv) constructs the graph induced by representatives of the partition. In the next section we show how to implement this operation in polylogarithmic parallel time. In section 4 we outline an algorithm for constructing the adjacency matrix of a cograph from its cotree representation. In the remaining part of this section we show that the length of the sequence $(G_0,F_0,U_0),...,(G_k,F_k,U_k)$ constructed using the reduce operation is O(log *n*) and that having this sequence we can construct the cotree $T_0$ in polylogarithmic parallel time.

Consider a leaf *u* of the cotree $T_i$. Let *r* denote a label. Let *u* be the representative of a set U in the partition $U_i$. The following diagram summarizes the substitutions of fragments for representatives (sometimes together with its parent) in the tree $T_i$ to obtain tree $T_{i-1}$. Their validity follows from lemmas 2 and 4.

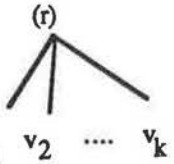
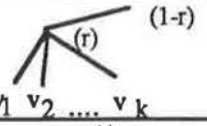
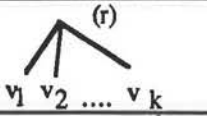
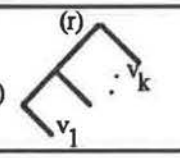
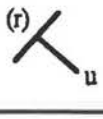
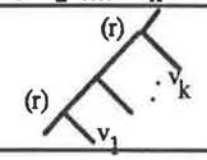
| type of set represented by u | fragment associated with it | the position of u in the cotree $T_i$ | possition of the fragment in the cotree $T_{i-1}$ |
|---|---|---|---|
| single vertex set |   |   |   |
| r-bunch set |   |   |   |
| | |   |   |
| r-line set |   |   |   |

Figure 10

To prove that the length of the sequence is O(log $n$), we note that the operation reduce satisfies the following properties:

**Property 3.** Consider the set B of vertices in the graph (i.e. leaves in the cotree) which are in bunch sets of the partition. After a single application of the reduce operation at most $\lfloor |B|/2 \rfloor$ of these vertices remain.

**Property 4.** Let the partition have $k$ line sets. Consider the set L of vertices in the graph which are in line sets of the partition. After a single application of the reduce operation exactly $k$ of these vertices remain.

These properties imply the following theorem :

**Theorem 1.** After O(log$n$) applications of the operation reduce a cograph is reduced to a single vertex.

**Proof :** It suffices to show that a single application of the operation reduce removes at least 1/10 of the current leaves.

Suppose that the operation reduce is applied to a cograph with $t$ vetices. Let B be the set of vertices in the cograph which are in bunch sets of the partition. Consider the following cases :

(1) $|B| \geq t/5$. Then, considering only leaves removed from bunch sets of the partition, the number of vertices left is less or equal to $t - |B| + |B|/2 = t - |B|/2 \leq 9/10\, t$.

(2) $|B| < t/5$. Let $k$ be the number of branching nodes in the cotree. Notice that $k \leq |B| - 1$ and the number of contractible sequences is at most $k$. Add the root to the set of branching nodes. With each branching node (except the root) we can associate a path

of internal nodes in such a way that the first node, say $v$, is a branching node and the last node is the closest ancestor of $v$ whose father is a branching node. For every such path there are at most four leaves which are children of nodes in the path and are not contractible leaves (see figure 11 for the worst case configuration). So after the application of reduce operation the number of leaves which are left is at most $|B|/2 + 4k + 1 \leq 9/2\,|B| - 3 < 9/10\,t$.

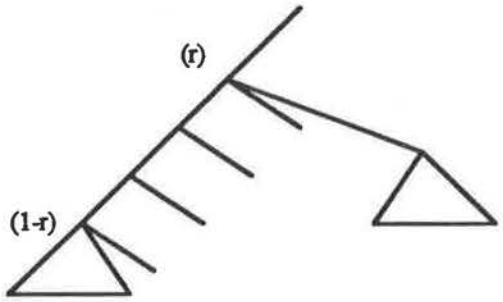



Figure 11

The algorithm outlined above assumed that the given graph was a cograph. It can be modified to work without this assumption as follows:

```
(* construct the sequence of triples (G_i,U_i,F_i) *)
    M := ⌈log_{9/10} n⌉
    i := 0;
    U_0 := { {v_k} | v_k ∈ V};
    for 1 ≤ k ≤ n do F_0^k = {v_k};
    while |U_i| > 1 and i ≤ M do
          i := i + 1;
          (* construct the next triple (G_i,U_i,F_i) *)
          reduce;                          --- see section 4 for details
          od;

    if i > M
          then the input graph is not a cograph;

          else      (* construct the cotree *)
              T_i := F_i^1;     (* there is only one fragment in F_i *)
              while i>0 do
                  i := i - 1;
                  obtain T_i by substituting for each representative of the partition U_{i+1} the
```

corresponding fragment from $F_{i+1}$
**od**

*(* check if correct *)*
Construct the cograph G' defined by cotree $T_0$; --- *see section 5 for details*
**if** $G' \neq G_0$ **then** the input graph is not a cograph.

## 4. Implementation of the operation reduce

The operation reduce finds the partition of the vertices of a graph into bunch sets, line sets and single vertices. It also identifies representatives of those sets and constructs corresponding fragments. We describe this operation in two phases. In the first phase, we define the main steps of the operation. In the second phase, we describe the implementation of those steps. We also note when to check conditions which might disqualify the input graph as a cograph. Some of the technical details of the implementation are left to the reader.

The operation reduce proceeds is follows :

1. For each pair of vertices $v,w$ check if $v$ and $w$ are siblings.
2. Find bunch sets, their representatives and construct corresponding bunches.
3. For each pair of vertices $v,w$ check for a vertex $u$ such that $Z_i(v,u,w)$ (i=0,1). Such a vertex $u$ is a contractible vertex. If there exists a vertex $x$ such that $Z_{1-i}(u,w,x)$ then $u,w$ are successive contractible vertices.
4. Find line sets, their representatives and construct corresponding lines.
5. Obtain $G_{i+1}$ by removing from $G_i$ all vertices not chosen as representatives.

The details of the implementation are as follows:

**Step 1.** This step can be implemented in O(log $n$) time with $n^3/\log n$ processors. For each pair of vertices $v,w$ compute the exclusive or of columns $v$ and $w$ of the adjacency matrix excluding rows $v$ and $w$ and then sum the elements of the resulting vertex. This can be done for each such a pair of vertices in O(log $n$) time with $n/\log n$ processors using the prefix sum computation algorithm described in [V84]. Store the results of this step in $n$ x $n$ array A by assigning $A(v,w) = 1$ if the resulting sum is zero (i.e. $v$ and $w$ are siblings) and $A(v,w) = 0$ otherwise. If there exist no sibling vertices (this can be checked in O(log $n$) parallel time) then the graph is not a cograph.

**Step 2.** Using the pointer hopping technique and the array A, each vertex can determine the vertex with the smallest index among its siblings. This can be done in O(log $n$) time with $O(n^2/\log n)$ processors. The unique vertex whose index is smaller than the index of its lowest indexed sibling is the representative of its bunch. The processor associated with this vertex determines its bunch-type (0-bunch or 1-bunch) and builds the parent node of the bunch. All the vertices in the bunch set construct pointers to the bunch parent ( whose address is known via the representative which is known to all the vertices in the bunch set). To allow the construction of the cotree a new copy of the representative is constructed along with a pointer its associated bunch. This copy participates in the next iteration.

**Step 3.** The implementation of this step is similar to that one of the step 1 but in this case we check if the corresponding prefix sum computation gives 1. If so, the position on which the difference occurs indicates the vertex $u$. Note that the relation $\Gamma(u) \neq \Gamma(v)$ has been checked in the previous step and that condition iii) can be checked in a constant time. Check first for the relation $Z_0$. Store the result in the array A by assigning $A(v,w) = u$ iff $Z_0(v,u,w)$ and $A(v,w) = 0$ if otherwise. It is possible for $A(v,w) = A(v',w') = u \neq 0$. However if the graph is a cograph then if $A(v,w) = u \neq 0$ and $A(v,w') = u' \neq 0$ then $u = u'$. For each vertex $u$ it can be determined if $u$ is an entry of A and, if so, a pair $(v,w)$ can be chosen such that $A(v,w) = u$. (This can be done in O(log$n$) time using a total of

$O(n^2/\log n)$ processors by exploiting the structure of A). If the pair $(v,w)$ is chosen for vertex $u$ then this is recorded in vector $B_0$ by setting $B_0(v) = u$ and $B_0(u) = w$.

In the similar way we can check for the relation $Z_1$. If for vertex $u$ the pair $(v,w)$ satisfying $Z_1(v,u,w)$ is chosen, then it is recorded in vector $B_1$ by setting $B_1(v) = u$ and $B_1(u) = w$.

**Step 4.** Note that $u_1,u_2$ are two successive contractible vertices iff there exists such vertices $v$ and $w$ such that $B_i(v) = u_1$, $B_i(u_1) = u_2$, $B_{1-i}(u_1) = u_2$ and $B_{1-i}(u_2) = w$. This follows from the fact that :

$$B_i(v) = u_1 \ , B_i(u_1) = u_2 \Rightarrow Z_i(v,u_1,u_2) \text{ and}$$

$$B_{1-i}(u_1) = u_2, B_{1-i}(u_2) = w \Rightarrow Z_{1-i}(u_1,u_2,w).$$

So we can construct table B such that $B(u_1) = u_2$ iff $u_1$ and $u_2$ are two successive contractible vertices. From this we can construct the corresponding contractible sequence. This step can be implemented in $O(\log n)$ time with $O(n^2/\log n)$ processors using standard pointer hopping techniques. As in the case of a bunch, we construct copies of representatives. Each copy keeps pointers to the beginning and to the end of its associated line.

Summarizing the discussion above, we have the following lemma:

**Lemma 6.** If the input graph is a cograph then we can construct its cotree in $O(\log^2 n)$ parallel time, using $O(n^3/\log n)$ processors.

## 5. Adjacency matrix construction from the cotree representation of a cograph

The algorithm to construct the adjacency matrix from the cotree representation of a cograph is based on property 2 and the idea of top-down tree computation [ADKP87]. This leads to O(log$n$) time and O($n^2$/log$n$) CREW PRAM algorithm using the optimal (but unpractical) list ranking algorithm of Cole and Vishkin [CV86a] or O(log$n$) time and O($n^2$) processors algorithm using standard list ranking algorithm.

Binarize the cotree T in such a way that newly introduced internal nodes have the same label as the node whose split led to the given node (see figure 12).

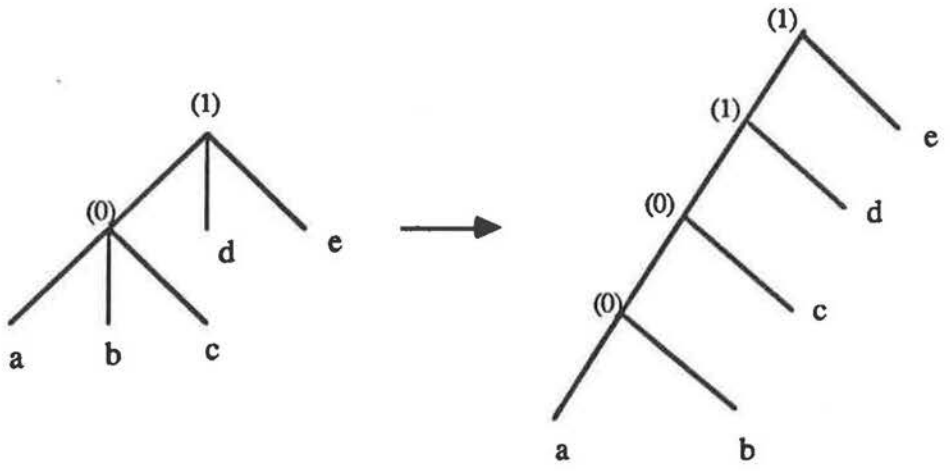



Figure 12

To construct i-th row of the adjacency matrix mark the path from i-th leaf to the root. Associate with each edge of the binarized cotree one a variable function. It is the constant function equal to the label of parent vertex for the given edge if the parent vertex is marked or the identity function otherwise. These functions define a decomposable top-down binary tree computation. As the result of this computation the value computed in a leaf j ($j \neq i$) is equal to one iff i,j are adjacent in the cograph represented by T. In [ADKP87] it is shown that such a computation can be performed in at most the time and processor cost of list ranking. To construct entire adjacency matrix we multiply the number of processors used by $n$ .

The construction above, together with lemma 6, implies the following lemma:

**Lemma 7.** We can test if an arbitrary input graph is a cograph and, if so, construct its cotree in $O(\log^2 n)$ parallel time, using $O(n^3/\log n)$ processors.

## 6.Reduction of the processor requirements

We can reduce the number of processors used by the algorithm to $O(n^3/\log^2 n)$ preserving $O(\log^2 n)$ time bound. Note that high number of processors in the implementation described above follows from the cost of computing relations $S_i$ and $Z_i$ (i=0,1). In each case the algorithm uses $n/\log n$ processors to compute each entry of the matrix A. Assume that we have only $n^3/\log^2 n$ to do the full job. Then some of the processors have to be involved in computing more then one entry to the array A (*cf.*steps 1,3 and 4 of the implementation of the operation reduce). With this number of processors we can still compute entries of an $n$ x $n/\log n$ submatrix using the method described in previous section. To reduce the number of processors divide matrix A to submatrices of the size $n$ x $n/\log n$ and compute elements of those submatrices one after another. Call columns of A which corresponds to removed vertices inactive columns. While the algorithm proceeds the number of active columns decrease. Our goal is to assign processors only to active columns. To do so keep a vector ACT of active columns and a variable NA equal to the number of active columns. Initially ACT(i) = i and NA = $n$. To compute elements of active columns of A, compute in parallel elements in first $n/\log n$ columns from the vector ACT then next $n/\log n$ columns, and so on. As shown in the proof of theorem 1, the number of vertices (and hence active columns) after i applications of reduce is at most

$(9/10)^i\, n$. Thus the time spent to compute entries to the matrix A by the cotree construction algorithm is

$$\sum_{i=0}^{M} \log n \lceil \log n \cdot (9/10)^i \rceil = O(\log^2 n)$$

It remains to show how to maintain the vector ACT. Assume that there is an additional vector ALIVE. After each application of the operation reduce ALIVE(i) = 1 if i is an active column and ALIVE(i) = 0 if otherwise. In order to determine for each active column its position in the vector ACT, it suffices to perform one prefix sum computation on elements in vector ALIVE.

## 7. Conclusion

With lemma 7 and the details of processor reduction described in section 6 we have completed the proof of our main result:

**Theorem 2.** We can test if an arbitrary input graph is a cograph and, if so, construct its cotree in $O(\log^2 n)$ parallel time, using $O(n^3/\log^2 n)$ processors.

The main difficulty in the efficient parallel construction of a parse tree for a cograph follows from the possibility of building an unbalanced cotree. Note that the same difficulty arises in the tree contraction problem ([ADKP87],[GR86],[MR85],[CV86b]) which is the basis for numerous parallel algorithms operating on trees. It appears that all efficient parallel algorithms for the tree contraction problem work in an iterative way. Typically in each iteration they remove leaves and shortcut long paths of the processed tree. In this paper this idea has been extended by applying it not just to operations on trees but also to the construction of trees themselves.

**Acknowledgment**

We are indebted to Derek Corneil for a number of helpful discussions on the results of this paper.

## References

[ADKP87] K. Abrahamsom, N. Dadoun, D.G.Kirpatrick and T.Przytycka. "A simple parallel tree contraction algorithm". Computer Science Department Technical Report 87-30, University of British Columbia, Vancouver, August 1987.

[B74] R. Brent. "The parallel evaluation of general arithmetic expressions". Journal of the ACM 21, 2, April 1974, pp. 201-206.

[CLS81] D.G. Corneil, H. Lerchs and L. Stewart. "Complement reducible graphs. " Journal of Discrete and Applied Mathematics 3, 1981, pp. 163-175.

[CV86a] R. Cole and U. Vishkin. "Approximate and exact parallel scheduling with applications to list, tree and graph problems. " In 27th Annual Symposium on Foundations of Computer Science, 1986, pp. 478-491.

[CV86b] R. Cole and U. Vishkin. "The accelerated centroid decomposition technique for optimal parallel tree evaluation in logarithmic time". Ultracomputer Note #108, TR-242, Dept. of Computer Science, Courant Institute NYU, 1986.

[GR86] A. Gibbons and W. Rytter. "An optimal parallel algorithm for dynamic tree expression evaluation and its applications". In Symp. on Foundations of Software Technology and Theoretical Comp. Sci., 1986, pp. 453-469.

[H86a] X. He. "Efficient parallel algorithms for solving some tree problems. " In 24th Allerton Conference on Communication, Control and Computing, 1986, pp. 777-786.

[H86b] X.He. "Parallel recognition and decomposition of two terminal series parallel graphs" , Computer & Information Science Research Center Technical Report, The Ohio State University Colunbus, 1986

[HCS79] D.S. Hirschberg, A.K. Chandra, D.V. Sarwte. "Computing connected components on parallel computers". Communication of ACM, August 1979, Vol 22, Number 8.

[L71] H. Lerchs. "On cliques and kernels. " Dept. of Comp. Science , University of Toronto, March 1971.

[MR85] G. L. Miller and J. Reif. "Parallel tree contraction and its application. " In 26th IEEE Symp. on Foundations of Computer Science, 1985, pp. 478-89.

[S78] L. Stewart. Cographs, A Class of Tree Representable Graphs. M. Sc. Thesis, Dept. of Computer Science, Univ. of Toronto, TR 126/78, 1978.

[V84] U. Vishkin. "Randomized speed-ups in parallel computation". In 16th Annual Symp. on Theory of computing, 1984, pp 230-239.