Generalized *PQ*-trees

Mark B. Novick*

TR 89-1074
December 1989

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

*Department of Computer Science, Cornell University, Ithaca, NY 14853-7501. Supported by NSF grant CCS-8806979.

Generalized PQ-trees

Mark B. Novick *

Abstract

We introduce a new data structure, which we call generalized PQ-trees because they behave like Booth and Lueker's PQ-trees. Given a ground set of n elements S, and $A = \{A_1, \ldots, A_k\}$ a collection of subsets of S, generalized PQ-trees allow us to efficiently represent which subsets of S never partially overlap with sets in A. We give an $O(kn)$ time sequential algorithm and an $O(kn)$ processor parallel algorithm for computing the generalized PQ-tree. Our new data structure can be used to speed up other researchers algorithms for recognizing interval and parity graphs.

*Department of Computer Science, Cornell University, Ithaca, NY 14853-7501. Supported by NSF grant CCS-8806979

1 Introduction

PQ-trees are a data structure developed by Booth and Lueker [1] to efficiently solve the *consecutive ones problem.* The problem is: *Given a ground set of n elements S and $A = \{A_1, \ldots, A_k\}$, a collection of subsets of S, does there exist a linear ordering λ of S such that every element of A appears as a consecutive subsequence of λ?* They applied this algorithm to the problems of interval and planar graph recognition. Klein [3] defined operations on PQ-trees that were suitable for use in parallel algorithms, and showed how to implement these algorithms efficiently.

We look at a related problem, the *trivial intersection problem.* Here the problem is: *Given a finite ground set S and a collection A of subsets over S, which subsets of S have a trivial intersection with every element of A?* Two sets P and Q have a nontrivial intersection (denoted $P \sim Q$) when they are neither disjoint nor is one a subset of the other one. We show that by generalizing the notion of a PQ-tree, we can easily represent those subsets of S which have a trivial intersection with every element of A. The concepts and algorithms developed by Booth and Lueker, and by Klein have natural analogues for generalized PQ-trees, hereafter called gPQ-trees.

In both PQ-trees and gPQ-trees there are two types of internal nodes, P-nodes and Q-nodes. Every internal node has at least two children. The children of each internal node of a PQ-tree are ordered. This is not true for gPQ-trees. The leaves of both kinds of trees are the elements of S. We will follow the traditional practice of depicting P-nodes by circles and Q-nodes by rectangles. In Figure 2.1 we depict a gPQ-tree T over the ground set $\{a, b, c, d, e, f\}$.

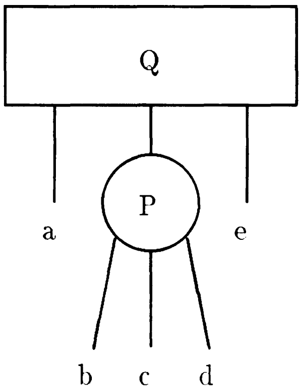


Figure 1: A gPQ-tree over the ground set {a,b,c,d,e,f} .

Each PQ-tree T represents the linear orderings of S consistent with the consecutive ones property. One such linear ordering may be found by reading off the leaves of T from left to right. Other linear orderings may be found by reading off the leaves from left to right after performing a combination of the following two operations:

1. The order of a children a Q-node can be reversed.

2. The children of a P-node can be arbitrarily permuted.

There is no way a PQ-tree over a nonempty ground set can represent the empty set of orderings. By convention we use a special *null* PQ-tree, denoted T_{null}, to represent the empty set.

To solve the consecutive ones problem, we let T be the universal PQ-tree in which every element of S is the child of a P-node which is the root of T. Every linear ordering of S is represented by the universal PQ-tree. We then *reduce* T with respect to A_1. The reduced tree represents all the linear orderings consistent with T in which the elements in A_1 also form a consecutive subsequence. We now let T be the reduced tree, and repeat the process by reducing it with respect to A_2, A_3, $\dots$, and finally A_k. If the null tree results, then the answer to the consecutive ones problem is NO. On the other hand if the resulting T is not null, then it represents the linear orderings consistent with the consecutive ones property. Booth and Lueker presented a sequential algorithm for performing the required reductions in $O(n + k + \mathrm{SIZE}(A))$ time where $\mathrm{SIZE}(A) = \sum_{1 \leq i \leq k} |A_i|$. Klein gave an NC algorithm which used $O(n + k + \mathrm{SIZE}(A))$ processors. He first showed how to perform the reductions when the A_i's are disjoint, and later solved the more general case where they need not be disjoint.

Each gPQ-tree T represents the subsets of S that have trivial intersections with every element of A. We denote this set of subsets by $M(T)$. The gPQ-tree is constructed so that the following subsets of S have trivial intersections with the elements of A:

1. The empty set, S, and all sets which contain only one element of S.

2. The set of leaves which are the descendants of some Q-node.

3. For any subset of children of a P-node, the set of leaves which are the descendants of this subset of children.

No null gPQ-tree is needed, since $\emptyset$, S, and all singletons have trivial intersections with A.

To solve the trivial intersection problem, we let T be the universal gPQ-tree in which every element of S is the child of a P-node which is the root of T. The universal tree represents the power set of S. We then reduce T with respect to A_1. The reduced tree represents the set of all subsets of S that are represented by T, and also $\not\sim A_1$. We now let T be the reduced tree, and repeat the process by reducing it with respect to A_2, A_3, $\dots$, and finally A_k. The resulting gPQ-tree represents all the subsets of S which are $\not\sim A_i$ for $i = 1, \dots, k$. See Figure 2.2 for an example.

We give a sequential algorithm for performing these reductions in $O(nk)$ time. We can convert this to an NC algorithm which uses $O(nk)$ processors. We solve the case

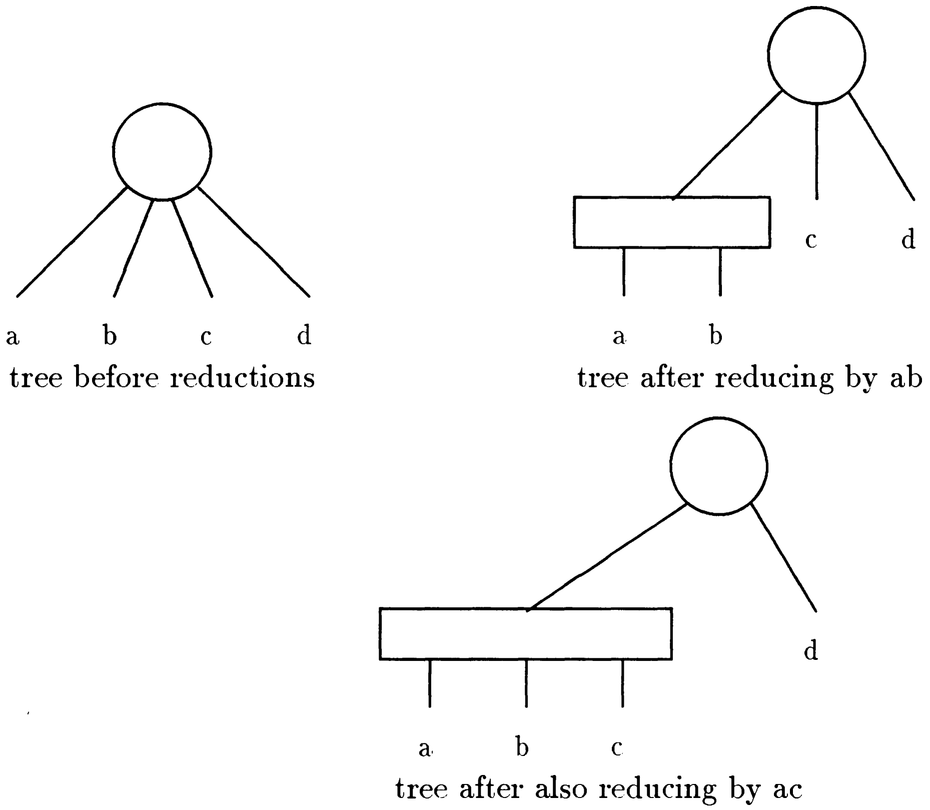


Figure 2: gPQ-tree reduction

where the elements of A are disjoint before moving on to the case where these elements need not be disjoint.

In the next section of the paper we extend concepts related to *PQ*-trees so that they now also apply to to *gPQ*-trees. We also prove some basic results about *gPQ*-trees in section 2. In section 3 we give an algorithm for the multiple disjoint reduction of *gPQ*-trees. This algorithm becomes the basis of the multiple nondisjoint reduction algorithm presented in section 4. In section 5 we show how to use *gPQ*-trees to solve some restricted versions of the trivial intersection problem. We conclude by showing how *gPQ*-trees can be used to find more efficient implementations of some existing graph algorithms.

2 Basic terminology and results

Definition 2.1 *If v is any node of some gPQ-tree T, the subtree rooted at v is itself a gPQ-tree, whose ground set is the set $leaves_T(v)$ of leaves that are descendants of v. We abbreviate this to $leaves(v)$ when it is clear which tree is being referred to.*

Definition 2.2 *Let A be a subset of the ground set S. We say a subset λ of S satisfies the set A if $A \not\perp \lambda$ i.e., $A \cap \lambda$ is equal to one of $\emptyset, A, or \lambda$. For a gPQ-tree T, let $\Psi(T, A) =$*

$\{\lambda \in M(T) : \lambda satisfies A\}$.

Theorem 2.1 *Given any T and $A \subseteq S$, there is a gPQ-tree $\hat{T}$ such that $M(\hat{T}) = \Psi(T, A)$, called the reduction of T with respect to A.*

Proof: We will show that the multiple disjoint reduction algorithm of section 3 can produce the reduction of T with respect to A.

We also define multiple gPQ-tree reduction. We define

$$\Psi(T, \{A_1, \ldots, A_k\}) = \{\lambda \in M(T) : \lambda \text{satisfies each} A_i (i = 1, \ldots, k)\}$$

In section 3, we give a parallel algorithm for multiple disjoint reduction, MDREDUCE($T, \{A_1, \ldots$ which modifies T to obtain a gPQ-tree $\hat{T}$ such that $M(\hat{T}) = \Psi(T, \{A_1, \ldots, A_k\})$ if the A_i's are disjoint.

Theorem 2.2 *MDREDUCE can be computed in $O(\log n)$ time using $O(nk)$ processors of a CREW PRAM.*

Using MDREDUCE as a subroutine, we give an algorithm, MREDUCE, that solves the more general reduction problem in which the sets A_i are not necessarily disjoint.

Theorem 2.3 *MREDUCE can be computed in $O(\log n \log(nk))$ time using $O(nk)$ processors of an "arbitrary" CRCW PRAM.*

Definition 2.3 *Let $lca_T(A)$ denote the least common ancestor of the leaves belonging to A. Suppose that $v = lca_T(A)$ has children. We say A is contiguous in T if*

1. *v is a Q-node, and for some subset C of the children of v, $A = \cup_{u \in C} leaves(u)$, or*

2. *v is a P-node or a leaf, and $A = leaves(v)$.*

The importance of a set being contiguous is illustrated by the following lemma.

Lemma 2.1 *Suppose that A is a nonempty subset of the ground set of the gPQ-tree T. Then A is contiguous in T if and only if each set $\lambda \in M(T)$ satisfies A.*

Proof: ($\Rightarrow$) Suppose A is contiguous in T, and $v = lca_T(A)$. If $\lambda \in M(T)$, then $lca_T(\lambda)$ is either an ancestor of v, a proper descendant of v, or neither. For these three cases the following holds respectively: $leaves(v) \subseteq \lambda$, $\lambda \subset leaves(v)$, or $leaves(v) \cap \lambda = \emptyset$. If v is a P-node, then $A = leaves(v)$ which implies that λ satisfies A. If v is a Q-node, we still have $A \subseteq leaves(v)$. We can still conclude in the first and third cases that λ satisfies A. If $lca(\lambda)$ is a proper descendant of v, and λ and A are not disjoint, then there exists a child v_i of v such that $\lambda \subseteq leaves(v_i) \subset A$. Thus λ satisfies A when v is a Q-node.

($\Leftarrow$) Assume every $\lambda \in M(T)$ satisfies A. If only one child v_i of $v = lca_T(A)$ satisfies the condition

$$leaves(v_i) \cap A \neq \emptyset$$

then $lca(A)$ is a descendant of v_i, contrary to the choice of v. Thus v has more than one child v_i that is not disjoint from A. Let C consist of those v_i's that are children of v which contain some element of A as a descendant. Suppose that for some child $v_i \in C$, we have $leaves(v_i) \not\subset A$. Then $leaves(v_i)$ does not satisfy A. Hence we have $leaves(v_i) \subset A$ for each $v_i \in C$. But then $A = \cup_{v_i \in C} leaves(v_i)$ because v was chosen to be an ancestor of every leaf in A. If v is a Q-node, this completes the proof that A is contiguous in T.

Suppose that v is a P-node, and v had some child v_i such that $leaves(v_i) \not\subset A$. We know that v has another child v_j that satisfies $leaves(v_j) \subset A$. Let $\lambda = leaves(v_i) \cup leaves(v_j)$. This set is a member of $M(T)$. However, λ does not satisfy A. Thus for every child v_i, we have $leaves(v_i) \subset A$. That is, $leaves(v) = A$. ∎

Lemma 2.2 *If the set λ satisfies two overlapping sets, it satisfies their union and intersection as well.*

Proof: We know that $\lambda \cap A$ is one of $\emptyset, \lambda, \text{or} A$, and $\lambda \cap B$ is one of $\emptyset, \lambda, \text{or} B$. By associativity, $\lambda \cap (A \cap B) = (\lambda \cap A) \cap (\lambda \cap B)$. This set equals

1. $\emptyset$ if either $\lambda \cap A = \emptyset$ or $\lambda \cap B = \emptyset$.
2. $A \cap B$ if either $A \subseteq \lambda$ or $B \subseteq \lambda$.
3. λ if both $\lambda \subseteq A$ and $\lambda \subseteq B$.

We cannot have $\lambda \cap A = \emptyset$ and $B \subseteq \lambda$ because we would then have

$$\emptyset = B \cap (\lambda \cap A) = (B \cap \lambda) \cap A = B \cap A \neq \emptyset.$$

By distributivity, $\lambda \cap (A \cup B) = (\lambda \cap A) \cup (\lambda \cap B)$. This set equals

1. λ if either $\lambda \subseteq A$ or $\lambda \subseteq B$.
2. $\emptyset$ if $\lambda \cap A = \emptyset = \lambda \cap B$.
3. $A \cup B$ if either $A \subseteq \lambda$ or $B \subseteq \lambda$.

∎

Corollary 2.1 *If A and B are overlapping sets that are contiguous in T, then their union and intersection are also contiguous in T.*

A reduction of a generalized PQ-trees agrees with one on the corresponding ordinary PQ-tree, assuming the ordinary tree does not become null. The gPQ-tree is still defined even when the ordinary PQ-tree becomes null. This is what makes them a true generalization.

Theorem 2.4 *If after reducing the universal PQ-tree by $A_1, \ldots, A_k$ we get a non-null PQ-tree T, then if we ignore the left-to-right order of the children in T we get the gPQ-tree that results from reducing the universal gPQ-tree by $A_1, \ldots, A_k$.*

Proof: ($\Rightarrow$) Let λ be a set that would satisfy every A_j if T describes a gPQ-tree. Let $v = lca_T(\lambda)$. Either $\lambda = leaves(v)$ or v is a P-node and λ is of the form $\cup_{v_i \in C} leaves(v_i)$, where C is some subset of the children of v. We will show that λ satisfies every set $A_j, 1 \leq j \leq k$. Let $u = lca_T(\lambda)$. If λ and A_j are disjoint, then A_j satisfies λ, so we will assume that they intersect. If either v is a Q-node and an ancestor of u, or v is a P-node and a proper ancestor of u, then $A_j \subseteq \lambda$. Booth and Lueker showed that for every every child u_i of u, either $leaves(u_i) \subseteq A_j$ or $leaves(u_i) \cap \subseteq A_j = \emptyset$. Furthermore, they proved that if u is a P-node, then $A_j = leaves(u)$. If $u = v$ is a P-node, then $\lambda \subseteq A_j = leaves(u)$. If u is a proper ancestor of v, then $leaves(v) \subseteq leaves(u') \subset A_j$, where u' is the ancestor of v that is also a child of u. In all cases λ satisfies A_j.

($\Leftarrow$) Suppose a set λ satisfies a A_j for j between 1 and k. Let $v = lca_T(\lambda)$. By an argument from the proof of Lemma 2.1, v has more than one child v_i that intersects λ. Let C be the set of these children. Suppose that for some child $v_i \in C$, we have $leaves(v_i) \not\subset \lambda$. Choose elements x, y, z satisfying $x \in \lambda \setminus leaves(v_i)$, $y \in \lambda \cap leaves(v_i)$, and $z \in leaves(v_i) \setminus \lambda$. There must exist some A_j which contains both y and z but not x because otherwise there would exist a satisfactory ordering containing the subsequence yxz. This A_j is not satisfied by λ. Hence we have $leaves(v_i) \subset \lambda$ for each $v_i \in C$. If v is a P-node, this completes the proof that $\lambda = \cup_{v_i \in C} leaves(v_i)$.

Suppose that v is a Q-node with children $v_1 \ldots v_s$ in left-to-right order. Let v_p be the leftmost child of v which contains an element of λ as a descendant. Let v_q be the rightmost such child. Since v is a Q-node, there are only two satisfactory orders of its children. In particular, $v_1 \ldots v_{p-1} v_q \ldots v_p v_{q+1} \ldots v_s$ is not a satisfactory order. The only way this order can be ruled out is if there is an A_j that contains both $leaves(v_{p-1})$ and $leaves(v_p)$ but not $leaves(v_q)$, or A_j contains $leaves(v_{q+1})$ and $leaves(v_q)$ but not $leaves(v_p)$. But $A_j \sim \lambda$. Thus for every child v_i, we have $leaves(v_i) \subset \lambda$. That is, $leaves(v) = \lambda$. ∎

3 Multiple disjoint reduction

In this section, we describe the algorithm MDREDUCE and prove its correctness, proving Theorem 2.2 and indirectly proving Theorem 2.1. We introduce some of Klein's terminology.

Definition 3.1 *Suppose that a node v of a PQ-tree has children $v_1 \ldots v_s$. To "insert" a node u between v and a consecutive subsequence $v_p \ldots v_q$ is to make the p-th child of v, and let the children of u be $v_p \ldots v_q$.*

In Figure 2.3 we show a P-node u being inserted between a P-node v and its children 2, 3, and 4. (In this figure and others to come, we use a triangle to represent a subtree; if the triangle is labeled, then we use the label to refer to the root of the subtree.)

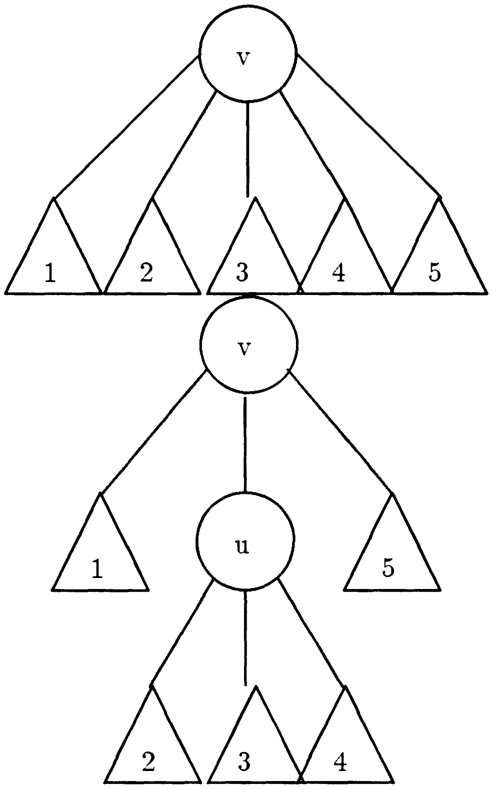


Figure 3: Inserting the node u between v and children 2, 3, 4.

We now consider the problem of reducing a gPQ-tree with respect to disjoint sets $A_1, \ldots, A_k$. For notational convenience, we summarize the disjoint sets in a "coloring" $\delta(v)$ of the ground elements $v \in S$, where the elements of each A_i receive a single color. We say that a set λ *satisfies* a color if the set of elements with that color has a trivial intersection with λ. We define

$$\Psi(T, \delta) = \{\lambda : \lambda \in M(T), \lambda \text{satisfies every color defined by} \delta\}$$

Note that since any ordering automatically satisfies a singleton set, we can assume every element of S receives a color – each element not appearing in any A_i receives its own unique color.

Our reduction algorithm consists of the following phases:

- **Pre-processing Phase:** The coloring δ of the ground elements is extended to a "coloring" Δ of all nodes of the gPQ-tree T.
- **Phase A:** P-nodes are processed and new nodes are inserted between each P-node and its children. The resulting gPQ-tree is denoted by T_A.
- **Phase B:** Q-nodes are processed. Certain sets of equivalence transformations are disallowed, by changing some Q-nodes into special nodes called *R-nodes,* to be defined later in this section. The resulting tree is denoted by T_B.

- **Post-processing phase:** Each of the R-nodes of the previous phase is eliminated and its children made children of its parent.

The structure of the proof is as follows: We first prove that $M(T) \supseteq M(T_A) \supseteq \Psi(T, \delta)$. It follows that $\Psi(T_A, \delta) = \Psi(T, \delta)$. Finally, we prove that $M(T_B) = \Psi(T_A, \delta) = \Psi(T, \delta)$. Thus T_B generates exactly the desired sets.

We first consider the pre-processing stage. This stage consists of extending the coloring δ of the leaves to obtain a coloring Δ of the entire tree.

The following terminology will be used throughout the proof.

Definition 3.2 *For an internal node v of T, we say a color is complete at v if all the leaves with that color are descendants of v. We say a color is incomplete at v if some, but not all, of the leaves of that color are descendants of v. We say that a color covers v if all the leaves below v are of that color, and that v is uncovered if no color covers v. Node v is orientable if there is some color that is incomplete at v, but this color does not cover v.*

In general, the coloring of the ground elements imposes constraints on the ordering of the children of each internal node u. If a color is complete at a child u of v, however, that color does not constrain the ordering of v's children at all. The constraints arise because of colors incomplete at children of v. Therefore, the first step in extending the coloring is to compute, for each internal node v, the set $INC(v)$ of colors incomplete at v.

Lemma 3.1 *For any color c we can determine the nodes it is incomplete at in $O(\log n)$ time using $O(n/\log n)$ processors. Since there are k colors, we can calculate $INC(\cdot)$ for all nodes in $O(\log n)$ time using $O(nk/\log n)$ processors.*

Proof: We use parallel tree contraction to compute several functions efficiently. Define $f(v) = 1$ if some leaf descendant of v receives color c. Otherwise, define $f(v) = 0$. Define $g(v) = 1$ if v contains two or more children v_i with $f(v_i) = 1$, otherwise set $g(v) = 0$. Both f and g can be evaluated by parallel tree contraction in $O(\log n)$ time using $O(n/\log n)$ processors. Find the node v closest to the root that satisfies $g(v) = 1$. We can compute the depth of every vertex and then sort the vertices with $g(v) = 1$ by their depth to do this. Only one such node exists, since if two of them did, their least common ancestor would also have a g value of 1. Color c is incomplete at every node u that is a proper descendant of v and satisfies $f(u) = 1$. Color c is not incomplete at any other node. ∎

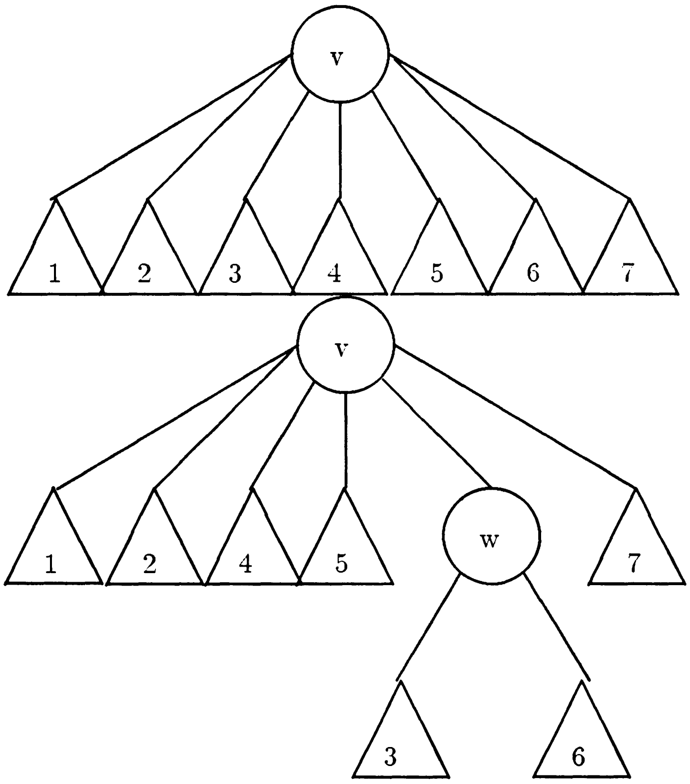
Phase A

1. Group together the children of v that are covered by the same color. For each color c, if there are at least two children covered by c and at least one child not covered by c, insert a new P-node w_c between these c-covered children and v.

2. Construct an auxiliary graph G_v whose nodes are the children of v, where for each color c, there is an edge between children v_i and v_j at which c is incomplete *if* v_i is the leftmost incomplete child. Identify the connected components of G_v. Group together the children of v by the connected component of G_v that they appear in. Insert a new Q-node between these nodes and v.

3. Let $S = \{v_i : v_i \text{a child of} v, INC(v_i) = \emptyset\}$. If every child of v is in S, then end. Otherwise, group together the children of v that appear in S, insert a new P-node $\hat{v}$ between v and the subset S (if $|S| > 1$), and rename v to be a Q-node.

The following figure shows Phase A being applied to a P-node.

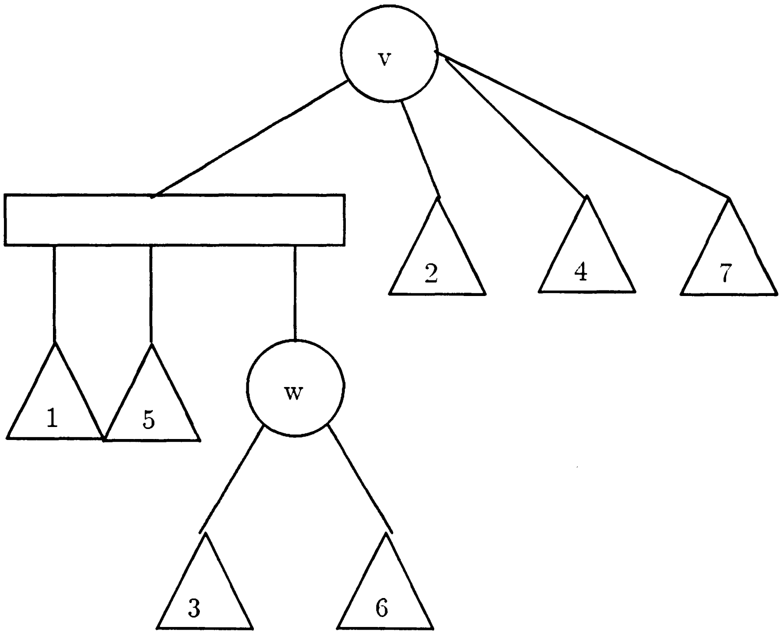


Phase A after step 1

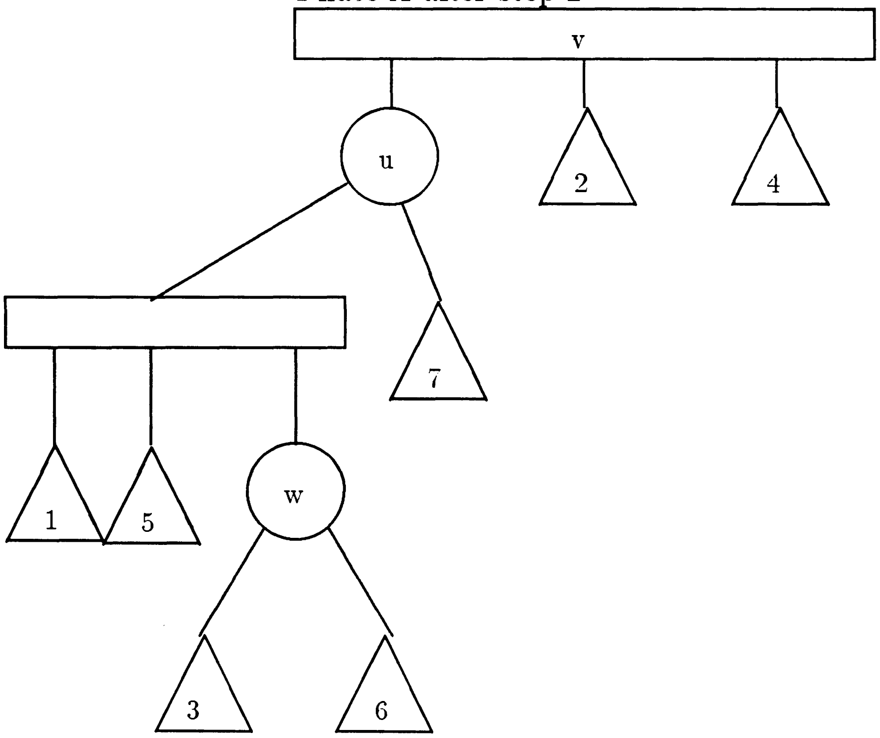
The coloring of the nodes is as follows: At node v tan and blue are the only incomplete colors. Tan is the only color incomplete at 2, but tan does not cover 2. Red is incomplete at 1 and 5. Blue is the only color incomplete at 4. Red is complete at 3 and 6. Child 7's descendants all have the same color, and this color does not appear elsewhere.

Lemma 3.2 *Phase A can be implemented in $O(\log n)$ time using $O(nk)$ processors of a CRCW PRAM.*

Proof: Determine which colors are incomplete at each node. Represent each of the k colors by a distinct integer from 1 to k. Each covered child with color $\chi(v_i)$ tries to write its node number into position $\chi(v_i)$ of a k element array. Only one child succeeds



Phase A after step 2



Phase A after step 3

Figure 4: Applying Phase A to P-node v, with children labeled 1 through 7. For this example, the colors are *blue*, *red* and *tan*.

in the case of a conflict. Make each node that did not succeed point to the node that succeeded. The connected components of the resulting forest consist of vertices with the same color. This algorithm uses $O(n)$ processors in $O(1)$ time so it can be slowed up to use $O(n/\log n)$ processors in $O(\log n)$ time.

If color c appears $l(c)$ times, then it introduces $l(c) - 1$ edges to the collection G_v graphs since each edge introduced separates the nodes with color c. Summing up over all colors we find that overall there are at most $n - k$ edges in the G_v graphs. We can find the connected components in an m vertex graph in $O(\log n)$ time with $O(m)$ processors by the algorithm of Shiloach and Vishkin [6]. With $O(kn)$ processors we can do all of these calculations. The grouping operations can be done like they were in the last paragraph. ∎

Lemma 3.3 *No P-node of T_A is orientable.*

Proof: Consider step 3. If the set S defined there contains all of v's children, then no color is incomplete at a child of v in T_A, and hence $INC(v) = \emptyset$, in which case v is not orientable. If S is not empty, then v is renamed a Q-node in step 3. A node w_c created in step 1 is covered by c. For a P-node $\hat{v}$ created in step 3, if u is a child of $\hat{v}$, then no color is incomplete at u. Hence no color is incomplete at $\hat{v}$. ∎

Lemma 3.4 $M(T_A) \subseteq M(T)$.

Proof: Note that for every node v of $T, leaves_{T_A}(v) = leaves_T(v)$. Moreover, every node v that is a Q-node in T is also a Q-node in T_A. Thus any set in $M(T)$ is also in $M(T_A)$. ∎

Lemma 3.5 *Let v_i, v_j, v_k be distinct incomparable nodes in gPQ-tree T. Let u be the least common ancestor of these three nodes with u_i, u_j, u_k being the children of u which are ancestors of v_i, v_j, and v_k respectively. (Two of these ancestors can be the same node.) Let x, y, z be elements of $leaves(v_i), leaves(v_j), leaves(v_k)$ respectively such that y and z both receive the same color c, but x does not. Then no set $\lambda \in \Psi(T, \delta)$ can contain x and y but not z. Furthermore, any $\lambda \in \Psi(T, \delta)$ must contain $leaves(u_i) \cup leaves(u_j) \cup leaves(u_k)$ if it contains both x and y.*

Proof: No $\lambda \in M(T)$ can contain x and y but not z because if it did then λ would have a nontrivial intersection with the set of elements of color c. Any $\lambda \in M(T)$ containing both x and y must also contain all of $leaves(u_i)$ because λ must consist of the union of all the leaf descendants of some node in T. By similar reasoning, λ also contains $leaves(u_j)$ and $leaves(u_k)$.

Lemma 3.6 $M(T_A) \supseteq \Psi(T, \delta)$.

Proof: Let v be a P-node of T and v_i, v_j, and v_k be some of its children. Assume $\Psi(T, \delta) \neq \emptyset$ since otherwise the lemma is trivial. By Lemma 3.5 any $\lambda \in \Psi(T, \delta)$ which contains $leaves(v_i)$, where v_i is covered by some color c, but also contains some element of a different color, then λ must also contain the leaves of every child of v covered by c. Hence in step 1 when a P-node w_c is inserted between these covered children and v, this transformation can be carried out needlessly eliminating sets from the gPQ-tree representation.

Next, let $\{v_i, v_j\}$ be an edge of the graph G_v constructed in step 2. By construction of G_v, both v_i and v_j contain a descendant with color c. If c covers one of these nodes, then it does not cover the other. By Lemma 3.5 any $\lambda \in \Psi(T, \delta)$ which contains $leaves(v_i)$ and $leaves(v_j)$ must contain the leaves of every child of v which contains an element with color c. Hence when we insert a new Q-node is inserted between v and the children in a connected component of G_v no sets are needlessly eliminated from the gPQ-tree representation.

After step 2 is performed every color c is incomplete at most one child of v. Suppose color c is incomplete at a child v' of v, and c does not cover v'. This happens because there is a leaf node that is not a descendant of v but has color c. Choose $v_i, v_j \in leaves(v')$ so that v_j receives color c, but v_i does not. Let v_k be a leaf node not in $leaves(v')$ with color c. As a consequence of the Lemma 3.5, if $\lambda \in \Psi(T, \delta)$ contains $leaves(v')$ it must also contain $leaves(v)$. By inserting a new P-node between v and the its children that are never incompletely colored, we do not needlessly restrict the class of sets represented by the gPQ-tree. This completes the proof. ▮

For phase B, we will find it useful to introduce a new kind of node, an *R-node*, which is like a Q-node, only more restrictive. An R-node may be viewed as a notational device for signifying that the R-node's children should become children of its parent node. An R-node can be eliminated and its children re-attached to its parent without disturbing their order; the resulting tree generates the same set of sets. We illustrate this modification in the following figure, where we signify that a node is an R-node by using two lines to connect it to its parent. R-nodes do not enhance the expressibility of PQ-trees. Phase B just consists of renaming each orientable node v and calling it an R-node.

Lemma 3.7 $M(T_B) = \Psi(T_B, \delta)$.

Proof: If v is an orientable node, then $leaves(v)$ cannot be an element of $Psi(T, \delta)$. This can be seen by applying Lemma 3.5. By making v into an R-node, we make the minimal change in the gPQ-tree implied by this observation. Every color c now induces a contiguous set in T_B so it can not reduced any further.

4 Multiple nondisjoint reductions

In this section, we describe how to perform multiple reductions on a gPQ-tree in parallel even when the sets we are reducing by are not disjoint. First, we describe some basic

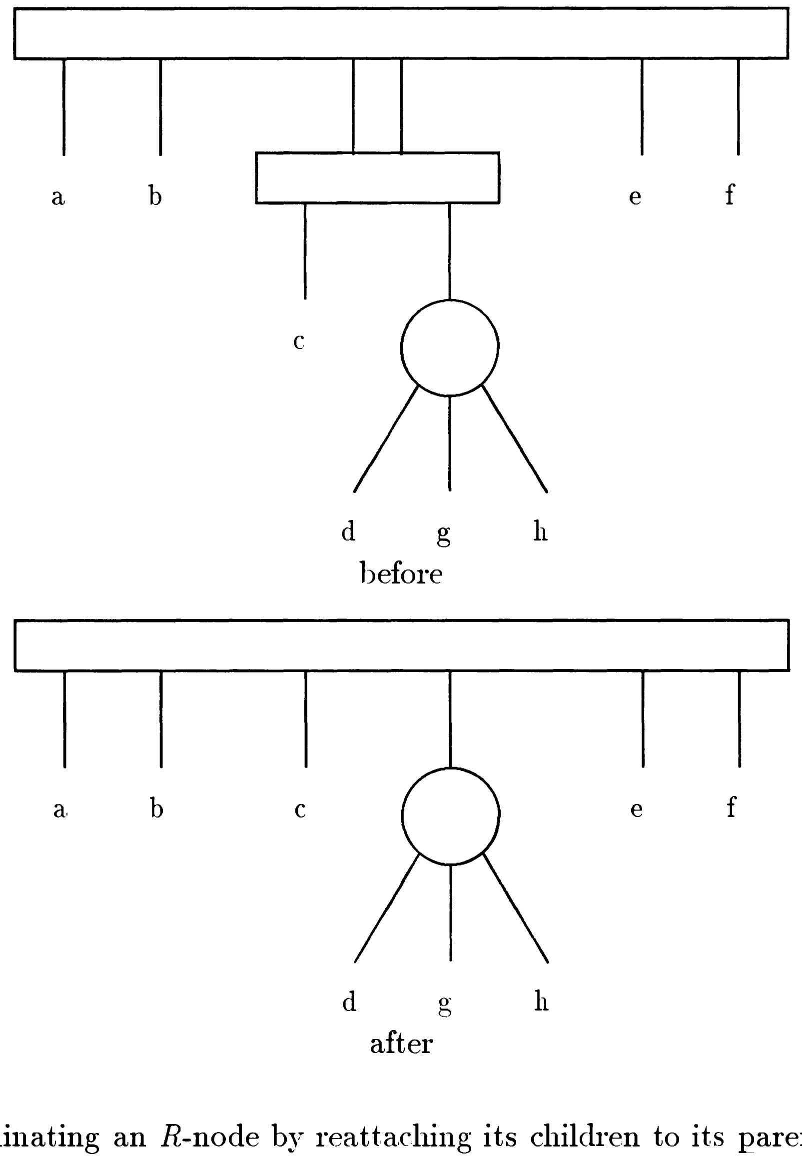


Figure 5: Eliminating an R-node by reattaching its children to its parent without disturbing their order

constructions on gPQ-trees that will be useful in this algorithm. We assume the ground set is S, unless specified otherwise.

4.1 gPQ-tree manipulations

Definition 4.1 *Given two sets $A, E \subseteq S$, we let $A|E$ denote $A \cap E$ and A/E denote $A \setminus E$ if $A \cap E = \emptyset$ or $A \subset E$. Otherwise, introduce a new element $\star_{|E} \notin S$, and define $A|E$ to be $(A \cap E) \cup \{\star_{|E}\}$. To define A/E in this case introduce two new elements $\star_{1/E}, \star_{2/E} \notin S$. If $E \subseteq A$, let A/E denote $A \cup \{\star_{1/E}, \star_{2/E}\} \setminus E$. If $A \sim E$, let A/E denote $A \cup \{\star_{1/E}\} \setminus E$.*

Once we have reduced a gPQ-tree with respect to a set E, Lemma 2.1 implies that E is contiguous in T, which means that

- either $E = leaves(lca(E))$, if $lca(E)$ is a P-node or a leaf,
- or else there is a subset C of the children of $lca(E)$ such that $E = \cup_{v_i \in C} leaves(v_i)$.

We need to separate the portion of the tree pertinent to E from the rest of the tree. We call this segregation:

Definition 4.2 *We say E is segregated in T if $E = leaves(lca_T(E))$.*

If E is contiguous in T, we can modify T, obtaining T' so that E so that E is segregated in T' and $M(T') = M(T)$. Namely, if E is not already segregated, then $v = lca_T(E)$ is a Q-node, by contiguity. Insert an R-node z between the children in C and v. In the resulting tree T', $z = lca_{T'}(E)$ and $leaves(z) = \cup_{v_i \in C} leaves(v_i) = E$. Moreover, it follows from the definition of the R-node that $M(T') = M(T)$. We can easily carry out this modification or undo it in $O(\log n)$ time and $n/\log n$ processors, where n is the number of nodes in T. In fact, if $E_1, \ldots, E_k$ are disjoint subsets of the ground set of T, each contiguous in T, we can segregate T with respect to all these sets within the same resource bounds.

Definition 4.3 *Suppose E is segregated in T. We let $T|E$ denote the subtree of T rooted at $lca_T(E)$. Thus $T|E$ is a gPQ-tree whose ground set is exactly E. We let T/E denote the gPQ-tree obtained from G by deleting all proper descendants of $lca_T(E)$. Thus if the ground set of T is S, the ground set of T/E is $S \setminus E \cup \{lca_T(E)\}$. Moreover, we shall typically consider the leaf $lca_T(E)$ as identified with two distinguished symbol $\star_{1/E}$ and $\star_{2/E}$.*

Definition 4.4 *Suppose E is segregated in T. If $lca_T(E)$ is an R-node, we say E is rigid in T. Otherwise, we say E is hinged in T.*

The following observation is useful to subsequent sections.

Observation 4.1 *Suppose the nonempty set E is satisfied by the set λ. Then for a set A of elements,*

1. *If $A \subseteq E, \lambda$ satisfies A if and only if $\lambda|E$ satisfies A.*

2. *If $A \cap E = \emptyset, \lambda$ satisfies A if and only if λ/E satisfies A.*

3. *If $A \supset E, \lambda$ satisfies A if and only if λ/E satisfies A/E.*

4. *Suppose $A \sim E$. Then λ satisfies A if and only if*

   (a) *$\lambda \subseteq E$ and either $\lambda \cap (A \cap E) = \emptyset$ or $\lambda \subset A \cap E$.*

   (b) *$\lambda \cap E = \emptyset$ and either $\lambda \cap (A \setminus E)$ or $\lambda \subseteq (A \setminus E)$.*

   (c) *or $E \subseteq \lambda$ and $A \setminus E \subseteq \lambda \setminus E$.*

4.2 Nondisjoint reduction algorithm

The approach we take to the problem of multiple nondisjoint reduction is as follows. Suppose we need to reduce the gPQ-tree T with respect to the sets $A_1 \ldots A_k$, called *reduction sets*. We choose a subset E of S that, according to Lemma 2.2, must be satisfied by every ordering satisfying $A_1, \ldots, A_k$. We reduce T by E, and then segregate E in T. We then recursively solve a multiple reduction problem on (possibly modified versions of) $T|E$ and T/E, and reattach the resulting gPQ-trees to obtain the reduced version of T. The key is in choosing a subset E at each level of the recursion in such a way that only $O(\log n)$ levels of recursion are needed.

We first describe the modified version of subtrees $T|E$ and T/E, and describe how reduction of T relates to the reduction of the subtrees. Then we review the notion of an intersection graph, and show how to compute a spanning forest for such a graph. Finally, we give the algorithm for nondisjoint reduction.

In Section 3, we described a decomposition of T into subtrees $T|E$ and T/E, where E is segregated in T. Recall that $T|E$ consists of the subtree of T rooted at $lca_T(E)$, and T/E consists of the subtree obtained from T by deleting all proper descendants of $lca_T(E)$. We now show that if A_i is a set that either contains E or is contained in it, then reduction of T with respect to A can be accomplished by an appropriate reduction of $T|E$ or T/E.

Lemma 4.1 *Let T be a gPQ-tree with ground set S in which the set E is segregated. Then*

1. *reduction of $T|E$ with respect to a subset A_1 of its ground set E has the same effect on T as reduction of T with respect to A_1;*

2. *reduction of T/E with respect to a subset A_2 of its ground set $S \setminus E \cup \{\star_{1/E}, \star_{2|E}\}$ has the same effect on T as reduction of T with respect to the set*

$$A = \begin{cases} A_2 \cup E \setminus \{\star_{1/E}, \star_{2/E}\} & \text{if} \star_{1/E}, \star_{2/E} \in A_2 \\ A_2 & \text{if} \star_{1/E}, \star_{2/E} \notin A_2 \end{cases}$$

Proof:

1. let $\hat{T}$ denote the tree T after the subtree $T|E$ has been reduced by A_1. By construction, a set λ belongs to $M(\hat{T})$ if and only if $\lambda \in M(T)$ and $\lambda|E$ satisfies A_1. But according to Observation 4.1, part 1, λ satisfies A_1 if and only if $\lambda|E$ does.

2. Denote by $\hat{T}$ the tree T after the subtree T/E has been reduced by A_2. By construction, a set λ belongs to $M(\hat{T})$ if and only if $\lambda \in M(T)$ and λ/E satisfies A_2. But according to Observation 4.1, parts 2 and 3, λ satisfies A_2 if and only if λ/E satisfies A.

▮

We also have to cope with sets A_i having nontrivial intersections with E. We shall assume for the construction that E is rigid in T. In the relevant situation, where there is a set $A_i \sim E$, the following lemma enables us to make the assumption with no loss of generality.

Lemma 4.2 *If $A \sim E$ and each is contiguous in the gPQ-tree T, then each is rigid in T.*

Proof: If A is hinged in T, then $A \in M(T)$. Since E is contiguous, it satisfies every element of $M(T)$. Therefore, it satisfies A, which contradicts the fact $A \sim E$.

Once T is reduced with respect to E and A, each of these two sets is contiguous in T. by Lemma 2.1, hence also rigid by Lemma 4.2.

For the following definitions and lemma, fix a *gPQ*-tree T with ground set S. Fix a subset E of S that is segregated and rigid in T. We construct two trees, $T||E$ and $T//E$, as follows. The root node of tree $T||E$ is a new node V_E. One of its children is $\star_{|E}$, the other is $lca_T(E)$, which becomes an R-node in this tree. The subtree rooted at $lca_T(E)$ is the same as it was in T. Tree $T//E$ looks the same as tree T except for the portion that is hanging from $lca_T(E)$. In $T//E$ this node has two children, $\star_{1/E}$ and $\star_{2/E}$. See the following figure.

Lemma 4.3 *Reducing T with respect to A has the same effect on T as reducing $T||E$ with respect to $A||E$, reducing $T//E$ with respect to $A//E$, and reattaching the reduced trees.*

Proof: The proof when $A \not\sim E$ is similar to the proof of Lemma 4.1. Here we consider what happens when $A \sim E$.

Let $\hat{T}$ be the result of reducing $T||E$ with respect to $A||E$, reducing $T//E$ with respect to $A//E$, and then reattaching these subtrees. Let $\lambda \in M(T)$. We want to determine whether $\lambda \in M(\hat{T})$. Note that we cannot either $\lambda \subseteq E, A \subseteq \lambda$ or $\lambda \cap E = \emptyset, A \subseteq \lambda$. In the former case, $A \subseteq E$ holds while in the latter, $A \cap E \subseteq \emptyset$ holds. Both contradict the fact that $A \sim E$. Furthermore, by symmetry, we cannot interchange A and E in the above statements.

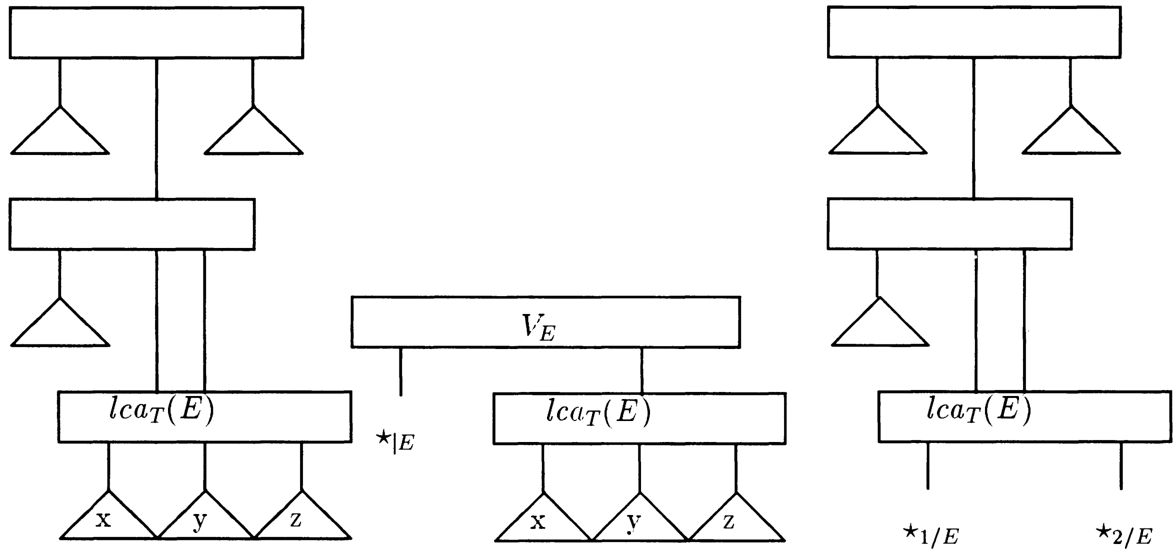


Figure 6: Constructing $T||E$ and $T//E$

We divide the analysis into three subcases, depending on how λ satisfies E. In the first subcase, $\lambda \subseteq E$. Then $\lambda \in M(\hat{T})$ is equivalent to λ satisfies $(A \cap E) \cup \star_{|E}$. This in turn is equivalent to saying that either these two sets are disjoint or λ is a subset of $A \cap E$. Thus λ satisfies A because $A \not\subseteq \lambda$.

In the second subcase, $\lambda \cap E = \emptyset$. Then $\lambda \in M(\hat{T})$ is equivalent to λ satisfies $(A \setminus E) \cup \star_{1/E}$. This in turn is equivalent to saying that either these two sets are disjoint or λ is a subset of $A \setminus E$. Thus λ satisfies A because $A \not\subseteq \lambda$.

Finally, if $E \subseteq \lambda$, then $\lambda \in M(\hat{T})$ is equivalent to $\lambda \cup \{\star_{1/E}, \star_{2/E}\}$ satisfies $(A \setminus E) \cup \star_{1/E}$. This in turn is equivalent to saying that $A \setminus E \subseteq \lambda \setminus E$. Thus λ satisfies A in this subcase too because $\lambda \cap A \neq \emptyset$ and $A \not\subseteq \lambda$. A family $\mathcal{F}$ of subsets of S defines a graph, the *intersection graph* of $\mathcal{F}$, whose vertices are the sets in $\mathcal{F}$, and where two sets are considered adjacent if they intersect. Let $\mathcal{H}$ be any subfamily of $\mathcal{F}$ whose intersection graph is connected; for example, we may take $\mathcal{H}$ to be a connected component of the intersection graph of $\mathcal{F}$. It follows from Lemma 2.2 that any subset of S satisfying each of the sets in $\mathcal{H}$ also satisfies the union $\mathcal{H}$.

To compute the connected components of (the intersection graph of) $\mathcal{F}$, we construct an auxiliary bipartite graph with vertex-set $\mathcal{F} \cup S$, where there is an edge between a set in $\mathcal{F}$ and an element of S if the element belongs to the set. Two sets in $\mathcal{F}$ are in the same connected component of the intersection graph if they are in the same component of the auxiliary graph. Moreover, a spanning forest of $\mathcal{F}$ can easily be obtained from a spanning forest of the auxiliary graph. Note that the number of edges in the auxiliary graph is just the sum of the cardinalities of the sets in $\mathcal{F}$.

We now give the algorithm for multiple nondisjoint reduction.

MREDUCE $(T, \{A_1, \ldots, A_k\})$

Let n be the size of the ground set of T. If $n \leq 12$, carry out the reductions one by one. Otherwise, let $\mathcal{A}$ be the family of sets $A_1, \ldots, A_k$. Let $\mathcal{F}$ consist of the sets A_i such that $|A_i| \leq n/2$. We call such sets "small." Let $\mathcal{D}$ be the remaining, "large," sets in $\mathcal{A}$. Find the connected components of the intersection graph of $\mathcal{A}$, find a spanning forest of the intersection graph of $\mathcal{F}$, and find the intersection $\cap\mathcal{D}$ of the large sets. There are four cases:

Case I: The intersection graph of $\mathcal{A}$ is disconnected. As observed above, it follows from Lemma 2.2 that for each connected component $\mathcal{C}_i$ of $\mathcal{A}$, a set satisfying $A_1, \ldots, A_k$ also satisfies $\cup\mathcal{C}_i$. Therefore, we reduce T with respect to the (disjoint) unions E_i of the connected components $\mathcal{C}_i$ of $\mathcal{A}$. Next, we consider each $T|E_i$ in parallel, and recursively reduce it with respect to the sets whose union is E_i. By Lemma 4.1, the effect on T is that of reducing it with respect to $A_1, \ldots, A_k$.

Case II: The union of sets in some connected component of $\mathcal{F}$ has cardinality at least $n/4$. In this case, we use a rooted spanning tree of the large connected component to find a connected subset of small sets whose union has cardinality between $n/4$ and $3n/4$. Fix any ordering $A_1, \ldots, A_s$ of the subsets in the spanning tree consistent with the distance from the root A_1. Any initial subsequence $A_1, \ldots, A_i$ is connected. Let $\hat{\imath}$ be the minimum i such that $\cup_{j=1}^{i} A_j$ has cardinality $\geq n/4$. Since each set is small, it follows that the cardinality of the union is no more than $3n/4$. We then let $E = \cup_{j=1}^{\hat{\imath}} A_j$. It follows from Lemma 2.2 that any ordering of S satisfying $A_1, \ldots, A_k$ also satisfies E. Therefore, we reduce T with respect to E. We proceed according to the following two subcase:

Subcase (1): There exists a set $A_i \sim E$. In this case, we further reduce T with respect to A_i. By Lemma 4.2, E is rigid in the resulting tree T. We can therefore decompose T into $T||E$ and $T//E$. We then recursively reduce $T||E$ with respect to $A_1||E, \ldots, A_k||E$, and reduce $T//E$ with respect to $A_1||E, \ldots, A_k||E$. We reattach the reduced trees. This effectively reduces T with respect to $A_1, \ldots, A_k$. by Lemma 4.3.

Subcase (2): Every $A_i \not\sim E$. In this case, we recurse on $T|E$ and T/E, reducing $T|E$ with respect to the sets $A_1|E, \ldots, A_k|E$, and reducing T/e with respect to the sets $A_1/E, \ldots, A_k/E$. We reattach the reduced trees. This effectively reduces T with respect to $A_1, \ldots, A_k$ by Lemma 4.1.

In each case, we precede the recursions by a step which discards reduction sets that are empty, full, or singleton; such sets have no effect.

Case III: The cardinality of the intersection of the large sets is at most $3n/4$. In this case, we can choose a subset $\mathcal{D}'$ of the large sets whose intersection has cardinality between $n/4$ and $3n/4$. To find $\mathcal{D}'$, order the sets in $\mathcal{D}$ arbitrarily, and choose the maximum i such that the intersection of the first i sets has cardinality at least $n/4$. Then $\mathcal{D}'$ consists of these first i sets. We let $E = \cap\mathcal{D}'$, and proceed as in Case II.

Case IV: None of the previous cases holds. In this case, we let $E = \cap \mathcal{D}$, and proceed as in Case II.

We now analyze the above algorithm. First, we show that only $O(\log n)$ levels of recursion are needed. Let n be the size of the ground set of T. Not that Case I cannot occur two times in succession. If Case II or III occurs, the ground sets of $T||E$ and $T//E$ each have size $\leq (3n/4)+2$. Suppose Case IV occurs. We assume that subcase (1) holds; the analysis for subcase (2) is identical. Because Case III does not hold, the size of the ground set of $T//E$ is at most $(n/4)+2$. Consider $T||E$ and the sets $A_1||E, \ldots, A_k||E$ with respect to which it must be reduced. By choice of E, the sets $A_i||E$ corresponding to large sets A_i are all full, *i.e.*, $E \subseteq A_i||E$, so we need only consider the collection $\mathcal{M}$ of sets $A_i||E$ corresponding to small sets A_i. Each connected component of the intersection graph of $\mathcal{M}$ has size less than $n/4$ because Case II does not hold. Therefore, although the ground set of $T||E$ might not be much smaller than that of T, after one further recursive call on $T||E$, the resulting gPQ-trees will each have ground sets of size at most $(n/4)+2$. Thus the algorithm terminates after $O(\log n)$ levels of recursion.

Next we consider the processor requirement. For a single level of recursion, we need k processors for each leaf of the gPQ-tree. This number of processors is sufficient to perform the gPQ-tree manipulations (*e.g.*, multiple disjoint reduction) and the intersection graph calculations (*e.g.*, spanning tree) for a single level of recursion. We also must verify that the recursive calls do not require many additional processors.

First we consider the number of gPQ-tree leaves. For Case I, the sum of the sizes of the ground sets of the $T|E_i$'s does not exceed the size of the ground set of T. For case II, III, and IV, the sum of the sizes of ground sets of $T||E$ and $T//E$ exceeds the size of the ground set of T by at most 3. Since there are only $O(\log n)$ recursions, one can show that the sum of the sizes of all ground sets at a given level of recursion is $O(n)$.

Next we consider the trees we recurse on. Suppose the top-level call to MREDUCE is

$$\text{MREDUCE}(\hat{T}, \{\hat{A}_1, \ldots, \hat{A}_k\}).$$

During the resulting execution, MREDUCE is recursively invoked many times on different gPQ-trees. At each level of the recursion, the sum, over all current invocations of MREDUCE, of the sizes of the trees is no more than a constant multiple of the size of $\hat{T}$ because constructing $T||E$ and $T//E$ adds only three new nodes, and we can do this at most n times.

We have shown that multiple nondisjoint reduction can be carried out in $O(\log n \log(nk))$ time using $O(nk)$ processors of a CREW PRAM. We have proved Theorem 2.3.

5 Restricted versions of the trivial intersection problem

In the trivial intersection problem we try to represent

$$\{\lambda : \lambda \sim A_i \text{for } i = 1, \ldots, k.\}$$

However, we could consider further restricting the problem by making λ satisfy either $\lambda \cap A_i = \emptyset$ or $\lambda \subseteq A_i$, for example. In this section we consider several such restrictions on each of the A_i's, and show how gPQ-trees can still be used to solve the problem. In each of these new versions λ still satisfies every A_i, so we first find the gPQ-tree which results from reducing by all the A_i's.

There are six versions that are important:

1. $\lambda \cap A_i = \emptyset$ or $\lambda \subseteq A_i$.

2. $\lambda \cap A_i = \emptyset$ or $A_i \subseteq \lambda$.

3. $\lambda \subseteq A_i$ or $A_i \subseteq \lambda$.

4. $\lambda \subseteq A_i$.

5. $A_i \subseteq \lambda$.

6. $\lambda \cap A_i = \emptyset$.

We mark the nodes of the PQ-tree with zeroes and ones. A zero indicates that the node does not correspond to a solution of the restricted problem, while a one says the opposite. After every node has been marked, we update the tree so that every node corresponds to a solution of the restricted problem.

To handle version 1, first find the least common ancestor (lca) of the leaf nodes of the gPQ-tree which contain vertices in A_i. Any proper ancestor of this node is marked 0. This lca node is also marked 0 if it properly contains A_i. All other nodes are marked 1. Version 2 is handled similarly. This time we mark all the proper descendants of the lca node with a 0, and all other nodes receive a 1.

We handle version 3 differently. Mark all leaf nodes containing a vertex in A_i with a 1. Mark all other leaves with a 0. Each internal node is marked with the OR of the marks its children receives.

Version 5 is a combination of the restrictions of versions 2 and 3. If a node is marked 0 in either of those two versions, then it gets marked 0 in version 5. Furthermore, $\emptyset$ is not a possible solution to the problem in this case. All other nodes are marked 1. Version 4 is a combination of the restrictions 1 and 3. Version 6 is a combination of the restrictions 1 and 2. Also if there is a node in the gPQ-tree whose leaf descendants are exactly A_i, then this node is marked 0. Again, all other nodes are marked 1. All the operations in the 6 versions can be done with $O(kn)$ processors in $O(\log n \log nk)$ time.

We can restrict λ with respect to different A_i's in different ways. For example, for some of them version 1 might apply, while in others version 2 does. If a node is ever marked 0 for any A_i, then mark it 0. Only mark it 1 if this is not the case. By looking at the nodes of the gPQ-tree marked 1, we can report all the sets which satisfy the restricted trivial intersections.

We can also take the OR of two restrictions and form a new restriction. For example, if A and B are sets, then we can have the restriction $A \subset \lambda \Rightarrow B \subseteq \lambda$. This is equivalent to $A \cap \lambda = \emptyset$ or $\lambda \subseteq A$ or $B \subseteq \lambda$, that is restriction 2 with respect to A or restriction 5 with respect to B.

We need to reduce the gPQ-tree so that nodes marked 0 are eliminated. The process for doing this is similar to Step 3 of Phase A of the disjoint reduction algorithm followed by Phase B (Q-node reduction.)

6 Applications to existing graph algorithms

The overlap of a family of sets is formed by constructing a graph whose vertices are sets in this family, and two vertices are connected by an edge if the corresponding sets have a non-trivial intersection. In both Przytycka and Corneil's [5] algorithm for parity graph recognition and Chen and Yesha's [2] algorithm for the consecutive ones problem, an overlap graph is explicitly constructed and its connected components found. Explicitly constructing the overlap graph appears to require $O(n^3)$ time sequentially, since the naive algorithm takes $O(n)$ time per vertex pair to check if the corresponding sets have a partial overlap. Generalized PQ-trees speed up this computation so that it runs in $O(n^2)$ time. If two vertices belong to the same connected component of the overlap graph, then the same node of the gPQ-tree is the least common ancestor of the elements of the corresponding sets. The bottleneck in both Przytycka and Corneil's algorithm and Chen and Yesha's algorithm was finding the connected components of the overlap graph so the use of gPQ-trees speeds up their algorithms.

Construction of an overlap graph is one case where the trivial intersection problem arises in practice. In another case, Novick [4] has shown that the split decomposition of a graph can be found quickly if the trivial intersection problem can be solved quickly. The use of generalized PQ-trees here results in efficient sequential and parallel algorithms for finding the split decomposition.

References

[1] K. S. Booth and G. S. Leuker. Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms. *J. Comput. Sys. Sci.*, 13:335–379, 1976.

[2] L. Chen and Y. Yesha. Efficient parallel algorithms for recognizing transformable convex bipartite graphs and finding their maximum matchings. Technical Report OSU-CISRC-8/88-TR26, The Ohio State University, 1988.

[3] P. N. Klein. Efficient parallel algorithms for planar, chordal, and interval graphs. Technical Report MIT/LCS/TR-426, Massachusetts Institute of Technology, 1988.

[4] M. B. Novick. Parallel algorithms for the split decompostion, 1990. forthcoming tech report.

[5] T. Przytycka and D. Corneil. Parallel algorithms for parity graphs. manuscript, 1989.

[6] Y. Shiloach and U. Vishkin. An $O(\log n)$ parallel connectivity algorithm. *J. of Algorithms*, 3:57–63, 1982.