An Efficient Parallel Algorithm for Planarity

by

Philip Nathan Klein
B.A., Harvard University
(1984)

Submitted to the Department of
Electrical Engineering and Computer Science
in Partial Fulfillment of the
Requirements of the Degree of

Master of Science in
Electrical Engineering and Computer Science
at the
Massachusetts Institute of Technology
June 1986

Signature of Author _____
Department of Electrical Engineering and Computer Science
May 16, 1986

Certified by _____
David Shmoys
Assistant Professor of Mathematics

Accepted by _____
Arthur C. Smith
Chairman, Departmental Committee on Graduate Students

An Efficient Parallel Algorithm for Planarity

by

Philip Nathan Klein

Submitted to the Department of
Electrical Engineering and Computer Science
on May 16, 1986
in Partial Fullfillment of the
Requirements of the Degree of

Master of Science
in
Electrical Engineering and Computer Science

Abstract We describe a parallel algorithm for testing a graph for planarity, and for finding an embedding of a planar graph. For a graph on n vertices, the algorithm runs in $O(\log^2 n)$ steps on n processors of a parallel RAM. The previous best algorithm for planarity testing in parallel polylog time ([Ja'Ja' and Simon, 82]) used a reduction to solving linear systems, and hence required $\Omega(n^{2.49\cdots})$ processors by known methods, whereas our processor bounds are within a polylog factor of optimal.

The most significant aspect of our parallel algorithms is the use of a sophisticated data structure for representing sets of embeddings, the PQ-tree of [Booth and Lueker, 76]. Previously no parallel algorithms for PQ-trees were known. We have efficient parallel algorithms for manipulating PQ-trees, which we use in our planarity algorithm. Our parallel PQ-tree algorithms also have other applications.

Thesis Supervisor: Dr. David Shmoys
Title: Assistant Professor

This thesis describes joint work with Professor John H. Reif. The research of the author was supported by an ONR Graduate Fellowship. The research of John Reif was supported by Office of Naval Research contract N00014-80-C-0647 and National Science Foundation Grant DCR-85-03251.

1 Introduction

1.1 Planarity

Planarity is a classical property of graphs, dating back to 1736, when Euler formulated the concept. A drawing of a graph on a plane in which no edges cross is called a *planar embedding.* A graph for which such an embedding exists is called a *planar* graph. The search for an efficient algorithm to decide planarity and find a planar embedding culminated in Hopcroft and Tarjan's linear-time algorithm ([10]).

Continuing in this tradition, we have developed a very efficient *parallel* algorithm for this problem. Our new algorithm finds a planar embedding for an n-node graph (or reports that none exists) in $O(\log^2 n)$ time using only n processors of an exclusive-write, concurrent-read P-RAM. Thus it achieves near-optimal speedup. The fact that only n processors are needed for our algorithm makes it feasible to implement on a real parallel computer.

In contrast, the previous best parallel algorithm for testing planarity, due to Ja'Ja' and Simon ([12]), reduced the problem to solving linear systems, and hence required $M(n)$ processors, where $M(n)$ is the number of operations required to multiply two $n \times n$ matrices. Ja'Ja' and Simon's algorithm was important because it showed that planarity could be decided quickly in parallel. However, such a large processor bound makes their algorithm infeasible. For any n large enough that a parallel algorithm would be preferred to the linear-time sequential algorithm (e.g., $n = 5,000$), $M(n)$ far exceeds the number of processors on any realistic parallel computer (e.g., $M(5,000) > 125,000,000,000$). Our planarity algorithm, on the other hand, would achieve the same time bound with only 5,000 processors.

In general, once a problem has been shown to be in NC, it is important for designers of parallel algorithms to look for *new* techniques which enable the problem to be solved using a minimum number of processors for the same parallel time bound. In particular, the number of processors should not exceed the sequential time bound for the same problem. Note that our new algorithm has this property. Realizing that any parallel algorithm requiring matrix multiplication or inversion to solve planarity would be unacceptably inefficient, we looked to completely different techniques. The inspiration for our parallel algorithm is a highly efficient *sequential* algorithm resulting from the combined work of Lempel, Even, and Cederbaum ([17]), Even and Tarjan ([7]), and Booth and Lueker ([4]). One essential ingredient we use from the work of Lempel, Even, and Cederbaum is that in building an embedding for a graph, premature committment to a particular embedding of a subgraph should be avoided. Instead, we use a data structure called a PQ-tree, due

to Booth and Lueker, to represent *all* embeddings of each subgraph.

Our parallel algorithm differs significantly from the sequential algorithm that inspired it. The sequential algorithm extended an embedding node by node. In contrast, we use a divide-and-conquer strategy, computing embeddings for subgraphs and combining them to form embeddings of larger subgraphs. To handle the numerous complications that arise in carrying out this approach, we are forced to generalize the approach of Lempel, Even, and Cederbaum.

In addition to being an historically important problem in algorithm design, finding a planar embedding is an essential ingredient in other algorithms. For example, the linear-time planar graph isomorphism algorithm of Hopcroft and Wong ([11]) started with planar embeddings of the input graphs. The linear-time algorithm of Lipton and Tarjan ([18]) for finding an $O(\sqrt{n})$ separator of a planar graph started with a planar embedding of the input graph. Our new linear-processor parallel algorithm for planarity has the following applications:

- planar graph isomorphism in $O(\log^3 n)$ time using n processors (using an algorithm due to Ja'Ja' and Kosaraju, which requires planar embeddings as input).
- finding a depth-first search tree of a planar graph in $O(\log^3 n)$ time using n processors (using ideas due to Justin Smith, combined with techniques of [23]).
- finding an outerplanar embedding of an outerplanar graph in $O(\log^2 n)$ time using n processors, and finding an $O(1)$ separator of an outerplanar graph.

Because our algorithm does not rule out any embeddings, it can be used to enumerate all possible planar embeddings of a graph at a rate of $O(\log^2 n)$ time per embedding, using n processors.

1.2 PQ-Trees

Our parallel planarity algorithm is rare among parallel algorithms in that it uses a sophisticated data structure. We have parallelized the *PQ-tree* data structure, due to Booth and Lueker ([4]), giving efficient parallel algorithms for manipulating PQ-trees. No parallel algorithms for PQ-trees existed previously. We define three operations on PQ-trees, *multiple-disjoint-reduction*, *join*, and *intersection*, and give linear-processor parallel algorithms for these operations. We use PQ-trees for representing sets of graph embeddings. However, PQ-trees are generally useful for representing large sets of orderings subject to adjacency constraints. Booth and Lueker use PQ-trees in efficient sequential algorithms for recognizing (0,1)-matrices with the consecutive one's property, and in recognizing and testing isomorphism of interval graphs. Using our parallel algorithms for PQ-trees, one

can recognize $n \times n$ (0,1)-matrices with the consecutive one's property in $O(\log^3 n)$ time using n^2 processors.

1.3 Outline of the Thesis

In Section 2, we describe some techniques useful for efficient parallel computation. In Section 4, we define PQ-trees and some operations on them, and describe parallel algorithms for these operations. In Section 5, we give some preliminary definitions and results concerning planar graphs. In Section 6, we describe in greater detail the planarity algorithm, using PQ-trees to efficiently represent sets of embeddings. We conclude in Section ??.

2 Efficient Parallel Computation

In this section, we discuss some techniques for efficient parallel computation, and give some applications of these techniques.

By "efficient parallel computation", we mean to suggest algorithms in which a single processor is assigned to each element of the input structure. For example, in our planarity algorithm, we assign a single processor to each node and each edge of the input graph. Moreover, our interest is in algorithms whose running times are polynomial in the logarithm of the size of the input structure, e.g. $O(\log n)$ or $O(\log^2 n)$.

Note: In Sections 4 and 6, we describe algorithms which have more than one processor per datum. However, the number of processors is always within a constant factor of the number of data. Note that we can always reduce a processor bound by a constant factor by using a single processor to simulate a constant number of others at the cost of multiplying the time required by a constant. For this reason, we typically state processor bounds for algorithm without using the "big-O" notation.

Our preferred model of computation is a P-RAM (see [8]), a parallel random-access machine, in which many processors are allowed simultaneous access to a common random-access memory. In the P-RAM, many processors are permitted to read the same memory location simultaneously, but no more than one processor is permitted to write to a single location. However, we will mention algorithms designed for a more powerful model of parallel computation in which concurrent writing is permitted; typically, conflicts are resolved by allowing exactly one of the processors attempting to write to a location to succeed.

2.1 Prefix Computation, Pointer-hopping, and Tree Contraction

Often a sequential computation can be formulated as follows:

> Given elements $x_1 \ldots x_n$ from some domain D and an associative operator $\circ$ on D, compute $x_1 \circ x_2 \circ x_3 \circ \cdots \circ x_n$.

For example, finding the product of n matrices can be so formulated.

A parallel solution to this problem is straightforward: form a balanced binary tree with the x_i's as leaves. Each internal node is viewed as a processor which applies the operator $\circ$ to the values computed by its children. The root of the tree will compute the value $x_1 \circ \cdots \circ x_n$. Such a tree has depth $\lceil \log n \rceil$, and therefore can compute the result in time $O(\lceil \log n \rceil)$, where we assume that $\circ$ can be applied to two arguments in contstant time.

A somewhat more general problem is:

> Given $x_1 \ldots x_n$ and $\circ$, compute each of $x_1 \circ x_2$, $x_1 \circ x_2 \circ x_3$, ..., $x_1 \circ \cdots \circ x_n$.

This lattter problem is called the *prefix* problem in [15]. The problem is trivial to solve sequentially. Indeed, the ordinary sequential approach to computing $x_1 \circ \ldots \circ x_n$ actually yields all the prefixes $x_1 \circ \cdots \circ x_i$.

A parallel solution to the prefix problem is less obvious. Fischer and Ladner give a construction for solving this problem in [15]. For pedagogical reasons, we give a solution that is more algorithmic than constructive.

Associate a processor π_i with each datum x_i. Let the processor π_i have a storage location y_i to hold a value of D. (For $j \leq 0$, let y_j denote the identify element for $\circ$). In the following algorithm, we use $y_i^{(t)}$ to denote the value of y_i at stage t.

1 Initially,

2 for each π_i in parallel $(i = 1, 2, \ldots, n)$,

3 let $y_i^{(0)} := x_i$

4 For $t := 0$ to $\lceil \log n \rceil - 1$ do

5 for each π_i in parallel,

6 let $y_i^{(t+1)} := y_{i-2^t}^{(t)} \circ y_i^{(t)}$

It is a simple induction to show that $y_i^{(t)} = x_{i-2^t+1} \circ \cdots \circ x_i$, where we let x_j denote the identity if $j \leq 0$. Hence $y_i^{(\lceil \log n \rceil)} = x_1 \circ \cdots \circ x_n$.

In the above algorithm, we assumed the data $\{x_i\}$ were arranged in an array. We can also solve the analogous problem in the context of linked lists, using essentially the same algorithm. Suppose we have n nodes that form a lined list. Associated with

each node v is a value $x(v) \in D$. Each node points to another node $p(v)$ to form the linked list—the last node in the list points to a dummy node v_{id} for which $x(v_{id}) = y(v_{id})$ =the identity for $\circ$. Let p^j be the j-fold composition of p with itself. To compute $x(v) \circ x(p(v)) \circ x(p(p(v))) \circ \cdots \circ x(p^{n-1}(v))$ for each node v, we assign to each node v a storage location $y(v)$ for storing an element of D, a storage location $q(v)$ for storing a pointer, and a processor $pi(v)$ to each node. Then we execute the following algorithm:

```
1    Initially,
2      for each π(v) in parallel,
3        let y^(0)(v) := x(v)
4        let q^(0)(v) := p(v)
5    For t := 0 to ⌈log n⌉ − 1 do
6      for each π(v) in parallel,
7        let y^(t+1)(v) := y^(t)(v) ∘ y^(t)(v)
8        let q^(t+1)(v) := q^(t)(q^(t)(v))
```

It is an easy induction to show that $q^{(t)}(v) = p^{2^t}(v)$ and $y^{(t)}(v) = x(v) \circ x(p(v)) \circ \cdots \circ x(p^{2^t}(v))$.

Observe that since our model of parallel computation permits simultaneous reads of the same location, we need not assume that the nodes v and pointers $p(v)$ form a list—indeed, they may form any directed acyclic graph with outdegree 1, i.e. a directed, rooted forest. For each node v, the algorithm will compute the $\circ$-product of the sequence of values x associated with all nodes from v to the root of the tree containing v. In this more general setting, this technique is typically called *pointer-hopping*; the operation $q^{(t+1)}(v) := q^{(t)}(q^{(t)}(v))$ is thought of as v "hopping" the node $q^{(t)}(v)$ to get to $q^{(t)}(q^{(t)}(v))$.

As an example of how this technique may be applied, let each $x(v)$ be 1, and let the operation $\circ$ be addition. Then the algorithm computes, for each node v, the depth of v in the tree containing it.

As a second application, we show how to preprocess an n-node tree T in $O(\log n)$ time so that subsequently, given any two nodes u and v, the least common ancestor of u and v in T may be found sequentially in $O(\log n)$ time.

We first compute a preorder numbering of the nodes of T (see, e.g. [28]). Tarjan and Vishkin describe a way of doing this using pointer-hopping. Each edge of the tree is replaced by two edges, one going down the tree and one going up. Using these edges, Tarjan and Vishkin obtain a linked list of the nodes of T in which each internal node appears twice and in which, excluding second appearances, the order is exactly the preorder. Using the previous example, we can compute, for each node v, the preorder number $pre(v)$

for each node v. We can similarly compute, for each node v the maximum preorder number $max(v)$ of a descendant of v. Now, given two nodes u and v, we can easily determine whether u is an ancestor of v. Namely, check if $pre(u) \leq pre(v) \leq max(u)$.

Next compute for each node v the nodes $p^1(v) = p(v), p^2(v), p^4(v), \ldots, p^{2^t}(v)$, and so on, up to $t = \lceil \log n \rceil$. Note that these are just the values $q^{(t)}(v)$ computed in the pointer-hopping algorithm. Let $p^0(v)$ denote v.

Now, suppose u and v are nodes in the tree T. To find the least common ancestor of u and v sequentially, we use a variant of binary search:

```
1   Initially,
2       let low := u
3       let h := 2^(⌈log n⌉-1)
4   While h > 0 do
5       if p^h(low) is not an ancestor of v then low := p^h(low)
6       h := ⌊h/2⌋
```

It is a simple induction to show that during the execution of the above algorithm, the least common ancestor of u and v is an ancestor of ℓow and a descendant of $p^{2h}(\ell ow)$, and that h is either a power of 2, or zero. ince h decreases by a factor of 2 at each iteration, there are $\lceil \log n \rceil - 1$ iterations. Upon termination, $h = 0$, so $\ell ow = p^{2h}(\ell ow) =$ the least common ancestor of u and v.

A problem closely related to prefix computation and its generalization to linked structures is called *tree evaluation*: Given a pointer structure forming a directed, ordered tree, where we assign a value to each leaf and a k-ary function f_v to each internal node v with k children, find the value at each internal node v resulting from applying f_v to the values of the children. When each node has at most one child, the structure is that of a linked list, and the problem reduces to the previous one.

The tree evaluation problem is trivial to solve sequentially: process the tree bottom up from leaves to root, applying each function to its arguments as they become available. Miller and Reif show how to solve this problem in parallel. For a tree with n nodes, their technique, *parallel tree contraction*, takes $O(\log n)$ time on a concurrent-write model of parallel computation. If we use an alternate representation in which the children of each node are arranged in a linked list, this same bound can be achieved on an exclusive-write model (see [21]).

Parallel tree contraction has that name because it can be viewed as a process of "contracting" a tree to a single node. (*Contraction* of an edge means identifying the edge's endpoints and removing the edge.) Tree contratction uses two operations, RAKE and COMPRESS. RAKE contracts all edges one of whose endpoints is a leaf. Thus,

RAKE essentially deletes all leaves. COMPRESS contracts edges between pairs of nodes each having one child. Thus contract will shorten a path by roughly a factor of two. Miller and Reif prove in [23] that, starting with an n-node tree, only $O(\log n)$ executions of RAKE followed by COMPRESS will contract the tree to a single node.

In tree evaluation, leaves are nodes for which a value is known. The RAKE operation corresponds to substituting the value of each leaf v into the function associated with the parent of v, hus obtaining a function with one fewer arguments. Suppose u is a node having a single child v and v also has a single child. The COMPRESS operation contracts the edge between u and v, identifying u and v. In tree evaluation, we assign to the resulting node the function obtained by composing f_u with f_v. The child of the resulting node is the previous child of v and the parent is the previous parent of u; note the resemblance to pointer-hopping.

The tree contraction process applied to tree evaluation computes the value for some of the original nodes of T, but not all. However, once tree contraction has terminated, it can be run "in reverse," expanding edges in the reverse order in which they were contracted. This process, callled parallel tree expansion, will allow a value to be assigned to each node in T. Thus, Miller and Reif's paralllel solution to the tree evaluation problem involves both a tree contraction phase and a tree expansion phase. However, we use "parallel tree contraction" to refer to the technique as a whole.

2.2 Sorting of Small Integers in Parallel

3 Introduction

There has been much work in the area of parallel algorithms for sorting (e.g. [3], [2], [16], [25]). However, much of this work has been in the tradition of comparison models, whereas in fact for many parallel algorithms in which sorting is a subroutine (e.g. the planarity algorithm of this thesis), it is sufficient to sort integers in the range $[0, \ldots, n^k - 1]$ for some constant k (we call them *small* integers). This is an easier problem. Indeed, while $\Omega(n \log n)$ is the lower bound for sequential time to sort n values in the comparison model, small integers can be sorted in $O(n)$ time on a unit-cost random access model, by radix sorting (see, e.g. [1]). One is therefore led tc investigate complexity bounds for parallel sorting of small integers.

In [24], Reif gives a randomized parallel algorithm for sorting small integers in $O(\log n)$ expected parallel time using $n/\log n$ processors of a concurrent-write model of parallel computation. Thus, Reif's algorithm is asymptotically optimal. One might wonder whether the randomness in Reif's algorithm can be eliminated without increasing the

complexity bounds by much. In fact, Reif states, "we would be quite surprised if any purely deterministic methods yield $PT = O(n)$ for parallel integer sort in the case of the bound $T = O(\log n)$. In this subsection, we describe a simple, deterministic parallel algorithm for small integer sorting. It runs in $O(\log n)$ time using n processors, so it is a factor of $\log n$ away from optimal. However, unlike the algorithm of [24], our algorithm runs on an exclusive-write model of parallel computation.

The algorithm is essentially an implementation of merge-sort. We represent the ordering on a set of elements by a modified binary trie[1] of height $O(\log n)$, whose leaves correspond to the keys. The procedure $TrieSort(i,j)$ will return a trie containing the elements $x_i, \ldots, x_j$.

$TrieSort(i,j) =$

If $i = j$ then return a trie containing only x_i
else compute in parallel
$T_1 = TrieSort(i, \lfloor \frac{i+j}{2} \rfloor)$ and $T_2 = TrieSort(\lfloor \frac{i+j}{2} \rfloor + 1, j)$
and return $Merge(T_1, T_2)$.

Having obtained the trie $TrieSort(1,n)$ for the entire sequence $x_1, \ldots, x_n$, we can easily obtain for each element its rank in the sorted order. First compute for each node v the number $lv(v)$ of leaves below v. This is done by a leaf-to-root pass of the tree in $O(\log n)$ time. Next, assign to each node the rank of the lowest-ranked leaf below it. In particular, each leaf is assigned the rank of the corresponding element. This is done by calling $rankTree(r,1)$, where r is the root of the trie and $rankTree$ is as below:

$rankTree(v,k) =$

assign k to v. If v has children v_1 and v_2, then
in parallel call $rankTree(v_1, k)$ and
$rankTree(v_2, k + lv(v_1))$.

It remains to describe the modified trie and the merging algorithm. To obtain a *stable* sort, we append the number i to the value of x_i as low-order bits. Then every value occurs exactly once. Let m be the length in bits of the resulting values.

Let $S \subseteq \{0, \ldots, 2^m - 1\}$. We will represent S by a tree $T(S)$ as follows. First, consider the complete binary tree with 2^m leaves. Label each node with a binary sequence which, when read left to right, describes the path from the root, where 0 means "go to the left child" and 1 means "go to the right child." Note that the leaves are labelled with m-bit representations of the numbers from 0 to $2^m - 1$, in left-to-right order. Mark each leaf whose number is in S, and then mark each node with some marked leaf as a descendent. Then the marked nodes form a tree. We contract long paths in this tree to single edges, so that the only nodes remaining are those with either no children (the leaves) or two children. This is $T(S)$. Note that because we have eliminated one-child

[1] See, e.g., [14].

nodes, the number of nodes is $2|S| - 1$. Note also that the height of the tree is no more than m. For a node v of the tree, let *v.label* be the binary string labelling v, and let $|v.label|$ be the length of the string (i.e. the depth of v in the complete binary tree). Note that if v and w are any two nodes (not necessarily in the same tree), the least common ancestor of v and w in the complete binary tree would be the node whose label is the initial substring (or *prefix*) common to *v.label* and *w.label*. In particular, v would be an ancestor of w in the full binary tree iff *v.label* is a prefix of *w.label*.

Each node v has three fields, *v.label*, *v.leftchild*, and *v.rightchild*. Given the roots r_1 and r_2 of two tries $T(S_1)$ and $T(S_2)$, respectively, and a variable *newroot*, $Merge(r_1, r_2, newroot)$ will first assign to *newroot* the root of a new trie $T(S_1 \cup S_2)$ and then proceed to compute the rest of the trie. It will be apparent to the reader that the merge can be carried out in time $O(\text{max. height of tries}) = O(\log n)$ where there is one processor for each node of the tries (i.e. $O(n)$). A naive implementation of *TrieSort* would therefore take time $O(\log^2 n)$, namely $O(\log n)$ per recursion for merging, times $O(\log n)$ stages of recursion. However, the reader should observe of the following algorithm that it processes the tries strictly from top to bottom, and can therefore be pipelined. That is, immediately after the root of the new trie is assigned to *newroot*, the new trie can begin to participate in another merge.

$Merge(r_1, r_2, \text{var}newroot) =$
    if neither of $r_1.label$ and $r_2.label$ is a prefix of the other, then
        let r be a new node.
        Set $r.label$ to be the common prefix of $r_1.label$ and $r_2.label$.
        Set $r.leftchild$ and $r.rightchild$ to r_1 and r_2
                in the proper order (lexicographic according to labels).
        Set $newroot$ to be v
    else
        assume that $r_1.label$ is a prefix of $r_2.label$ (else swap r_1 and r_2).
        Set $newroot$ to be r_1.
        If $r_1.label = r_2.label$ then
            Call
              $Merge(r_1.leftchild, r_2.leftchild, r_1.leftchild)$ and
              $Merge(r_1.rightchild, r_2.rightchild, r_1.rightchild)$ in parallel.
        else
            Let $i = |r_1.label|$.
            If $i + 1^{\underline{st}}$ bit of $r_2.label$ is 0 then
                $Merge(r_1, leftchild, r_2, r_1.leftchild)$
            else
                $Merge(r_1.rightchild, r_2, r_1.leftchild)$
            end if
        end if

    end if
end Merge.

4 Our Parallel PQ-Tree Algorithms

4.1 PQ-tree Definitions

In our planarity algorithm, we will need to represent large sets of sequences of sets of edges. These sets are too large to represent explicitly, so we make use of an efficient data structure, the PQ-tree, due to Booth and Luecker ([4]). In this section, we define the PQ-tree and some operations on it, and show how these operations may be carried out efficiently in parallel. We freely adapt the terminology of [4] to suit our needs.

A PQ-tree over the ground set S is a tree with two kinds of internal nodes, P-nodes and Q-nodes. Every internal node has at least two children, so the number of internal nodes is no more than the number of leaves. The children of each internal node are ordered. The leaves are just the elements of S. For example, in Figure 1 is depicted a PQ-tree T over the ground set $\{a, b, c, d, e, f\}$. Here, as henceforth, Q-nodes are depicted by rectangles and P-nodes are depicted by circles. Throughout this section, n will denote

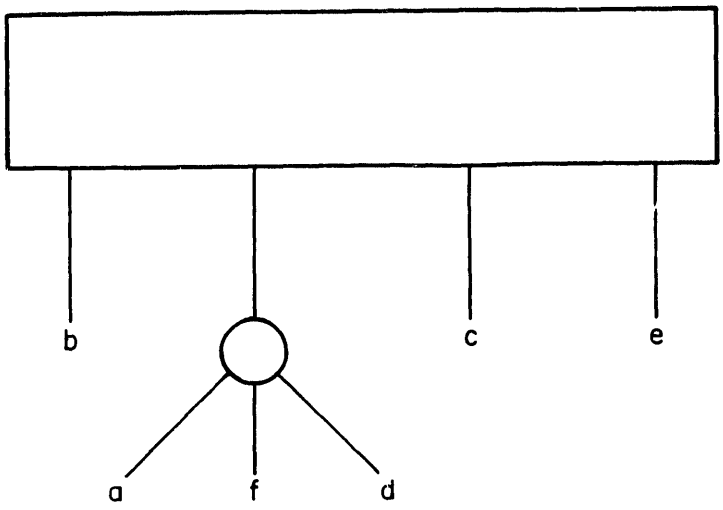

Figure 1: A PQ-tree over the ground set $\{a, b, c, d, e, f\}$.

the cardinality of S.

For concreteness, we will assume that a PQ-tree is represented by a pointer structure as follows: each node has a pointer to its parent, its left sibling, its right sibling, its leftmost child, and its rightmost child (using null pointers where necessary). This representation permits constant time insertion and deletion of consecutive sequences of children.

A PQ-tree is used to represent certain classes of linear orderings of its ground set S. Let T be a PQ-tree over S. We will denote by $L(T)$ the set of linear orders represented by T, and say that T *generates* $L(T)$. One element of $L(T)$ is obtained by reading off the leaves left to right in the order in which they appear in T. This is called the *frontier* of T, and written $fr(T)$. (The frontier of the tree in Figure 1 is *bafdce*.) The other elements are those linear orders obtained in the same way from all trees T' *equivalent* to T. We say T' is *equivalent* to T if T' can be transformed into T by a sequence of transformations. The permissible equivalence transformations are:

(1) the order of children of a Q-node may be reversed, and

(2) the children of a P-node may be arbitrarily reordered.

The first is called *flipping* a Q-node, and the second is called *reordering* a P-node. It is useful to think of a PQ-tree T as a representative of all the trees equivalent to it. We write $T' \cong T$ if T' is equivalent to T.

Note that for a node with only two children, the only possible reordering of its children is a reversal. Thus if a node has two children, it makes no difference whether it is a P-node or a Q-node. We shall automatically consider any node with two children to be a

Q-node.

Consider once again the PQ-tree T in Figure 1. There are 12 orderings in $L(T)$, including *bafdce*, *bdafce*, *ecafdb*, *ecfadb*. The first is just the frontier of T. The second ordering, *bdafce*, is obtained by reordering the P-node's children; the third ordering, *ecafdb*, is obtained by reversing the order of the Q-node's children; the fourth ordering, *ecfadb*, is obtained by both reversing the order of the Q-node's children and reordering the P-node's children.

Since there is no way a PQ-tree over a non-empty ground set can represent the empty set of orderings, we use a special *null* tree, denoted by T_{null}, to represent this empty set.

If v is any node of some PQ-tree T, the subtree rooted at v is itself a PQ-tree, whose ground set is the set $leaves_T(v)$ of leaves below v. Say that v *generates* the set of linear orderings of $leaves(v)$ generated by the subtree rooted at v. The *frontier* of v (in T) is just the frontier of the subtree of T rooted at v. We write $leaves(v)$ for $leaves_T(v)$ when T is implicit.

Note: Throughout this section, we use the terms *descendent* and *ancestor* to refer to non-proper descendents and ancestors, unless otherwise specified. That is, v is its own descendent and its own ancestor.

Let A be a subset of the ground set S. We say a linear ordering $\lambda = s_1 \ldots s_n$ of S *satisfies* the set A if all the elements of A are consecutive in λ, i.e. for some i and j, $s_i s_{i+1} \ldots s_j$ are all the elements of A. The *reduction* of the PQ-tree T with respect to A is a PQ-tree $\hat{T}$ generating exactly those linear orderings generated by T that satisfy A. We say of $\hat{T}$ that it is *reduced* with respect to A. Booth and Lueker prove that given any T and $A \subseteq S$, there is a reduction $\hat{T}$ of T with respect to A; in fact, they give an algorithm $REDUCE(T, A)$ which modifies T to get $\hat{T}$. Their algorithm works in time proportional to the cardinality of A_1. Note that if *no* ordering generated by T satisfies A, the reduction $\hat{T}$ is just the null tree.

In a typical algorithm using PQ-trees, one starts with a simple PQ-tree T and sequentially reduces it with respect to each subset in a sequence $A_1, \ldots, A_k$ of subsets. Each reduction further restricts the set of represented orderings. Note that by the definition of reduction, the final resulting PQ-tree $\hat{T}$ does not depend on the order of the subsets; $\hat{T}$ generates all orderings generated by T that satisfy every A_i. This observation suggests the existence of a parallel algorithm for computing $\hat{T}$ by performing all the reductions simultaneously on the single tree T.

Consider the special case in which the A_i's are all disjoint (we consider the general case below). In this special case, it is convenient to think of the A_i's as defining a coloring of some of the elements of the ground set, where an element v in A_i receives the color c_i. We say that an ordering *satisfies* the color c_i if all the c_i-colored elements occuring in the

ordering form a contiguous subsequence of the ordering. Then the intent of reduction is to introduce just enough new constraints into a PQ-tree T so that every color is satisfied by every ordering generated by the resulting PQ-tree. Note that a color that is assigned to only one element is trivially satisfied. Hence it does no harm to assign a distinct color to every element not appearing in any subset A_i. We may therefore assume that every ground element receives a color. Let the coloring be $cl : S \rightarrow C$, where C is the set of colors. In Subsection 4.2, we give a parallel algorithm MULTIPLE-DISJOINT-REDUCE(T, cl) which reduces T with respect to the coloring cl.

Theorem 4.1 *MULTIPLE-DISJOINT-REDUCE can be computed in $O(\log n)$ time using n processors.*

More generally, let $T_1, \ldots, T_k$ be PQ-trees over the ground sets $S_1, \ldots, S_k$, and suppose each ground element occurs in the ground sets of at most two trees. Let $A_{\{i,j\}} = S_i \cap S_j$ for all $1 \leq i, j \leq k$ $(i \neq j)$. For each tree T_i, the non-empty sets $A_{\{i,j\}}$ are all disjoint, so T_i can be reduced with respect to these sets. Carrying out all reductions simultaneously is called *all-adjacent-pairs reduction.*

Corollary 4.1 *All-adjacent-pairs reduction can be carried out in $O(\log n)$ time using n processors, where n is the total number of ground elements.*

We next define a new operation on PQ-trees, not considered in [4]. A PQ-tree T is the *intersection* of two PQ-trees T_1 and T_2 over the same ground set if $L(T) = L(T_1) \cap L(T_2)$. In Subsection 4.4, we describe an algorithm INTERSECT(T_1, T_2) for intersecting two PQ-trees using disjoint reduction as a subroutine. The algorithm modifies T_1 to be the intersection of the two original trees. Let n be the size of the ground set.

Theorem 4.2 *INTERSECT can be computed in $O(\log^2 n)$ time using n processors.*

Note that, like reduction, intersection can "fail," i.e. the result may be the null tree.

As an illustration of the usefulness of INTERSECT for parallel algorithms, we use it to obtain a parallel algorithm for reducing a PQ-tree T with respect to a sequence $A_1, \ldots, A_k$ of subsets that are not necessarily disjoint. The algorithm works in $O(\log^2 n \cdot \log k)$ time using $O(n^2)$ processors. Make k copies $T_1, \ldots, T_k$ of the PQ-tree T, and in parallel reduce each T_i with respect to A_i. Next, use a parallel prefix computation on the T_i's with the operation being intersection. Parallel prefix computation works in $\log k$ steps each of which takes $O(\log^2 n)$ time (to do the intersection). The parallel prefix computation gives us k trees: the reduction of T with A_1, the reduction of T with A_1 and A_2, the reduction of T with A_1, A_2, and A_3, and so on.

For an ordering λ of the elements of S satisfying a subset $A \subset S$, let $\lambda|A$ denote the induced ordering on the elements of A, and let $\lambda|\overline{A}$ denote the induced ordering on the

elements of $S - A$, i.e. the ordering obtained from λ by deleting elements of A. Let λ^R denote the reverse of λ.

Suppose we have two PQ-trees T_1 and T_2 over the respective ground sets S_1 and S_2. Let $A = S_1 \cap S_2$, and suppose T_1 and T_2 are reduced with respect to A. For $\lambda_1 \in L(T_1)$ and $\lambda_2 \in L(T_2)$, let λ_1 *join* λ_2 denote the ordering over $S_1 \cup S_2 - A$ obtained by substituting $\lambda_2|\overline{A}$ for $\lambda_1|A$ in λ_1. Let the join of T_1 with T_2 be the PQ-tree T generating

$$\{\lambda_1 \textit{ join } \lambda_2 \mid \lambda_1 \in L(T_1), \lambda_2 \in L(T_2), \text{ and } \lambda_1|A = (\lambda_2|A)^R\} \tag{1}$$

(The term "join" is inspired by the join of relational database theory.)

In Subsection 4.3, we show how to compute the join of T_1 with T_2 in the special case where T_2 is reduced with respect to $S_2 - A$. The join may be the null tree (in the event (1) is empty). Assuming that this is not the case, we can compute the join in $O(\log n)$ time (better still, $O(1)$ time after an $O(\log n)$ time preprocessing phase for a special case). However, to determine whether (1) is empty requires intersection, and hence takes $O(\log^2 n)$ time.

In Subsection 4.5, we discuss the representation of certain sets of cyclic orderings by PQ-trees. This subsection is directed specifically towards the application of PQ-trees to the parallel planarity algorithm. We give conditions under which reduction and join can be applied to PQ-trees representing cyclic orderings.

4.2 Reduction

In this subsection, we describe the algorithm MULTIPLE-DISJOINT-REDUCE. Before considering the most general case, let us consider a few easy examples. The tree in Figure 2 consists of a single P-node, all of whose children are leaves. This tree allows all possible orderings of its ground set. Now suppose we want to reduce the tree with respect to the set $\{c, e, f\}$. We want to permit all possible orderings in which these letters are consecutive. Think of first choosing any ordering of $\{c, e, f\}$, say fce, and then treating this as a single element $[fce]$ and choosing any ordering of the set $\{a, b, [fce], d, g, h\}$. The corresponding PQ-tree is depicted in Figure 3.

Continuing one step further, the reduction of the tree in Figure 2 with respect to the sets $\{c, e, f\}$ and $\{a, g, h\}$ is shown in Figure 4. Note that we would have obtained the same tree (up to equivalence) if we had started with the tree in Figure 3 and reduced it with respect to $\{a, g, h\}$, or if we had first reduced the tree in Figure 2 with respect to $\{a, g, h\}$ and then reduced the result with respect to $\{c, e, f\}$. It is apparent from the definition we gave for reduction that it is commutative; i.e. the result of reducing a tree with respect to a collection of subsets does not depend on the order in which the subsets are handled.

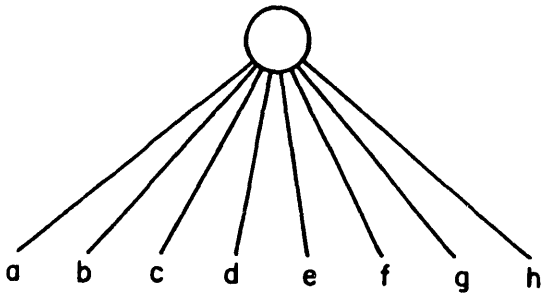

Figure 2: This tree permits all orderings of its ground set $\{a, b, c, d, e, f, g, h\}$.

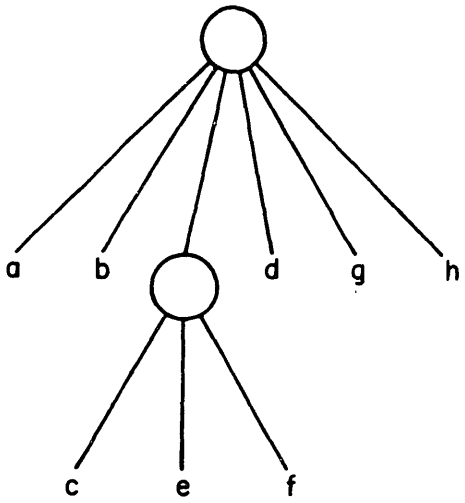

Figure 3: This tree permits those ordering of its ground set in which the elements of $\{c, e, f\}$ are consecutive.

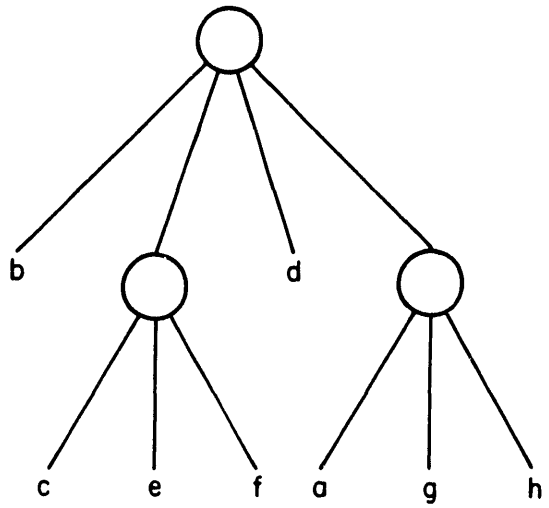

Figure 4: This tree permits those ordering of its ground set in which the elements $\{c, e, f\}$ are consecutive and the elements of $\{a, g, h\}$ are consecutive.

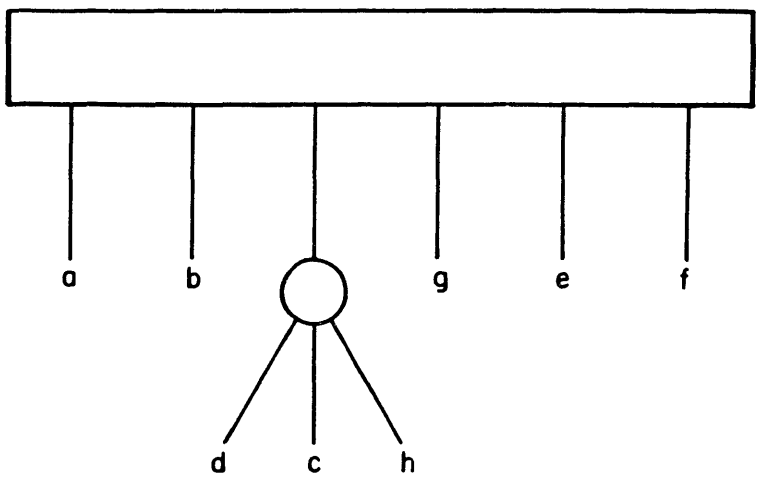

Figure 5: A PQ-tree over the ground set $\{a, b, c, d, e, f, g, h\}$.

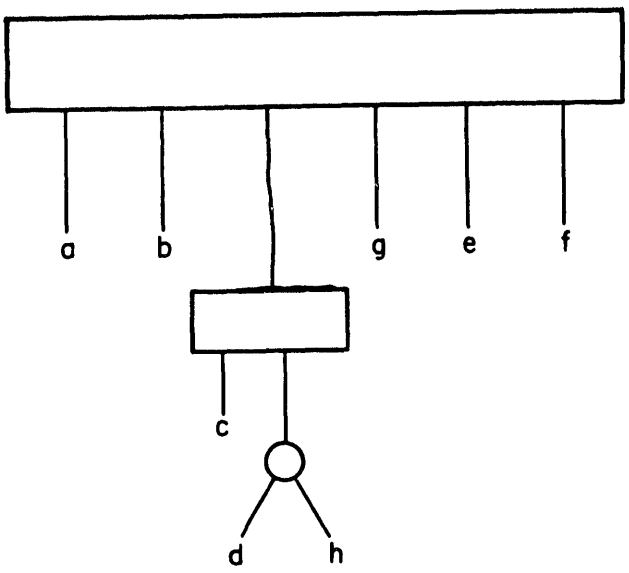

Figure 6: A first (and unsuccessful) attempt to reduce the PQ-tree of Figure 5 with respect to $\{a, b, c\}$.

So far, we have not considered the interactions arising when we reduce with respect to overlapping sets. The following example illustrates the sort of interactions that arise. Consider the PQ-tree in Figure 5, and suppose we want to reduce it with respect to the subset $\{a, b, c\}$. We check that a and b are already adjacent; we need not change the PQ-tree in order to bring them together. However, c hangs below a P-node permitting the arbitrary reordering of d, c, and h. We only want to permit those reorderings in which c ends up adjacent to a. An initial step would be to separate c from d and h, as in Figure 6. But while this does ensure that c is an *endpoint* (a leftmost or rightmost element) of the subsequence consisting of c,d, and h, it does not ensure that c will end up adjacent to a. The reason is that the parent of a and the parent of c are independently "flip-able." For example, by flipping the root (the parent of a) of the PQ-tree in Figure 6, we obtain a tree having a frontier in which a is not adjacent to c. We need to introduce a constraint requiring that when the parent of a flips, so does the parent of c, and vice versa.

In order to be able to express this constraint it is useful to introduce a new kind of node, an *R-node*. An R-node "follows the lead" of its parent node, in that, under any equivalence transformation, an R-node is flipped if and only if its parent is flipped. (An R-node is not permitted to be the child of a P-node.) To complete the previous example, then, we let the parent of a be an R-node, as depicted in Figure 7; we signify that a node is an R-node by drawing two lines connecting it to its parent.

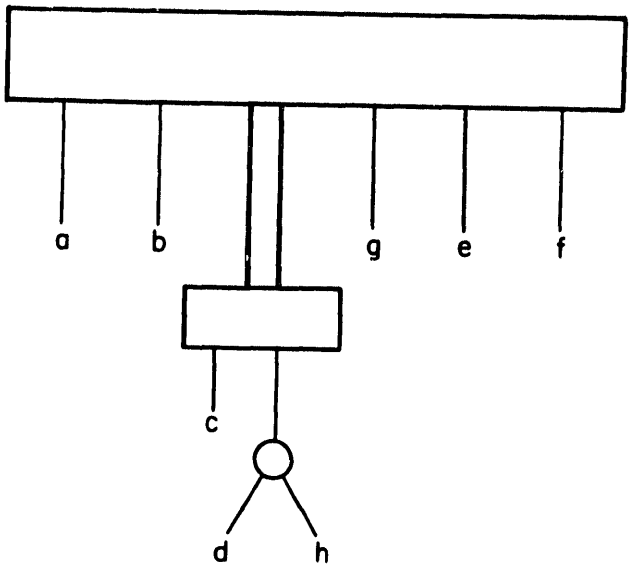

Figure 7: The reduction of the tree of Figure 5 with respect to $\{a, b, c\}$.

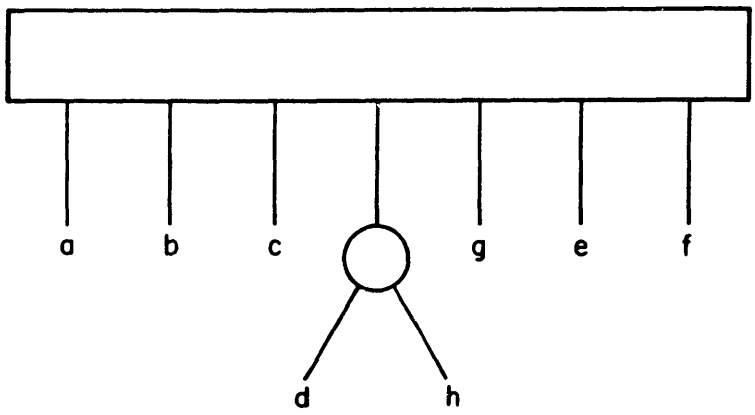

Figure 8: A PQ-tree equivalent to the tree of figure 7, but without an R-node.

Another way to view an R-node is as a notational device for signifying that the R-node's children should be inserted into the sequence of its parent's children. An R-node may be eliminated and its children reattached to its parent without disturbing their order; the resulting tree generates exactly the same set of orderings. For example, in Figure 8, we illustrate the result of eliminating the R-node from the PQ-tree of Figure 7. Thus R-nodes are merely a notational and computational convenience—they do not enhance the expressibility of PQ-trees.

As it turns out, the general case of disjoint reduction can be obtained by a modest generalization of these examples. The essential idea is that we operate on each node with regard to its children almost as if its children were leaves. The biggest difference

between an internal node and a leaf in this context is that an internal node has (at least) two different "orientations"—flipped or not flipped. Thus at times we will need to fix one orientation of an internal node with respect to its parent in order to effect the reduction, as in the last example. We introduced the R-node in order to be able to fix one orientation. After the reduction has been completed, we can eliminate the R-nodes.

The proof of Theorem 4.1 follows.

proof of Theorem 4.1: The algorithm for disjoint reduction consists of three stages:

(1) a pre-processing stage in which colors are assigned to the internal nodes of the PQ-tree in accordance with the coloring of the leaves,

(2) a transformation stage in which the tree structure is modified to obtain the reduced tree, and

(3) a post-processing stage in which R-nodes are eliminated.

The post-processing stage is the simplest, and we describe it first.

Lemma 4.1 *From any PQ-tree T with R-nodes, we can in $O(\log n)$ time using n processors obtain a PQ-tree T' equivalent to T such that T' has no R-nodes.*

proof First compute a preorder numbering of the tree (using techniques in [28]). Now, by use of parallel pointer-hopping techniques, each child v of an R-node can determine its lowest Q-node ancestor. This will be v's new parent. Each Q-node can use the preorder numbering to sort all its new children to obtain the proper order for them. The resulting tree is equivalent to the original tree, but has no R-nodes. □

We next consider the pre-processing stage. This stage consists of extending the coloring of the leaves to the entire tree.

The following terminology will be used throughout the proof. For an internal node v of T, say a color is *complete* at v if all the leaves with that color are descendents of v. Say a color is *incomplete* at v if some, but not all, of the leaves of that color are descendents of v. 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.

In general, the coloring of the ground elements imposes constraints on the ordering of the children of each internal node u. However, if a color is complete at a child v of u, that color does not constrain the ordering of u's children at all. The constraints arise because of colors incomplete at children of u. Therefore, we assign to each internal node v all the colors incomplete at v. Thus an internal node may have more than one color—but not many more, as the following lemma and its corollary show.

Lemma 4.2 *Suppose the ordering λ is generated by some node u of T and satisfies a color c which is incomplete at a descendent v of u. Let τ be the ordering obtained by restricting λ to leaves(v). Then c is the color of an endpoint of τ.*

proof of lemma The c-colored leaves are consecutive in λ, as are the leaves below v. Some c-colored leaves are below v and some are not, hence at least one endpoint of τ must be colored c. □

Corollary 4.2 *If any ordering generated by T satisfies all colors, then for each internal node v of T, there are at most two colors incomplete at v.*

The following lemma is analogous, but considers the endpoints of unicolored sequences.

Lemma 4.3 *Let v' be an uncovered node at which color c is incomplete. Then for any ordering λ generated by T, an endpoint of a c-colored consecutive subsequence of λ belongs to leaves(v').*

proof The leaves below v' occur as a consecutive subsequence τ in any ordering generated by T. Since τ does not consist entirely of leaves colored c, there must be two consecutive leaves only one of which is colored c. That is, one of the leaves below v' is an endpoint of a sequence of c-colored leaves. □

Corollary 4.3 *If any ordering generated by T satisfies all colors, every internal node has at most two uncovered children with the same color.*

proof Let v be an internal node, and consider any uncovered child v' with color c. By Lemma 4.3, an endpoint of a c-colored subsequence is a leaf with v' as an ancestor. But in any ordering satisfying c, the c-colored leaves form a single consecutive subsequence, and so there are only two endpoints. Thus v can have at most two uncovered children v' with color c. □

For node v, let INCOMPLETE(v) be the set of colors incomplete at v. (Ordinarily INCOMPLETE(v) will contain no more than two elements.)

Lemma 4.4 *If any order generated by T satisfies all colors, INCOMPLETE can be computed for all nodes v simultaneously in $O(\log n)$ time using n processors.*

proof By Corollary 4.2, we may assume that INCOMPLETE(v) never contains more than two elements. Consider the sequence of leaves of T, read left to right. For each color c, consider the first and the last node in this sequence that have color c. Their lowest common ancestor, which we will call LCA(c), is the lowest node in the tree at which the color c is complete. Using parallel tree contraction and expansion, it is easy to compute lowest common ancestors for all colors simultaneously within the stated bounds.

If v is a leaf colored c, INCOMPLETE(v) is either $\{c\}$, or, if v is the only node colored c, then INCOMPLETE(v) is empty. If v is an internal node, then

$$INCOMPLETE(v) = \left(\bigcup_{u \text{ is child of } v} INCOMPLETE(u) \right) - \{c : LCA(c) = v\}$$

For an internal node v, INCOMPLETE(v) can be computed by determining the colors incomplete at the children of v, and checking each such color c to see if LCA(c) is v. By Corollary 4.2, each child can be assumed to contribute at most two colors. Moreover, the colors contributed by all children can be checked against LCA($\cdot$) simultaneously. Using parallel tree contraction, we obtain the desired result. □

If at any node, the number of incomplete colors turns out to exceed two, the processor at that node should halt. After the computation completes, it can be determined in $O(\log n)$ time whether any processor has halted. If so, the result of the reduction can be taken to be T_{null}. Having checked the tree T for violations of Corollary 4.2 and computed INCOMPLETE($\cdot$), we can check T for violations of Corollary 4.3. For each internal node v, and each color c incomplete at a child of v, we can in parallel count the number of uncovered children of v with color c. If the number exceeds two, the processor at v should halt.

This completes the description of the pre-processing stage. We commence the description of the second stage: obtaining the reduced tree. The second stage consists of two phases, phase A and phase B. In phase A, each node v ensures that the ordering of its children is consistent with the coloring (possibly by adding new nodes), and assigns colors to temporary storage locations associated with itself and its children. In phase B, the values of the fields are processed, and the order of children of some nodes is reversed. Then some nodes are renamed to be R-nodes. The resulting PQ-tree generates exactly those orderings in $L(T)$ that satisfy all colors. The description of the second stage is rather detailed, and we recommend the reader skip it upon first reading.

The next two lemmas identify those trees equivalent to T whose frontiers satisfy all colors. Let T' be any PQ-tree derived from T by equivalence transformations. Consider the nodes of T' to be colored as they were in T. For each node v, let $\ell_{T'}[v]$ and $r_{T'}[v]$ be the leftmost and rightmost elements, respectively, of the frontier of v in T'. (We write $\ell[v]$ and $r[v]$ when T' is implicit.) It is a consequence of Lemma 4.2 that if the frontier of T' satisfies every color, then

$$\text{for each node } v, \text{ INCOMPLETE}(v) \subseteq \{cl(\ell[v], cl(r[v]\}. \tag{2}$$

Lemma 4.5 *Assume (2) holds for T'. For each node u, the following two conditions are equivalent:*

(i) The frontier of u in T' satisfies every color.

(ii) For each (non-proper) descendent v of u, if $v_1 \ldots v_s$ are the children of v in order, then $\ell[v_1]r[v_1]\ell[v_2]r[v_2]\ldots\ell[v_s]r[v_s]$ satisfies every color.

proof by induction on height of u. Trivial for u of height 0, for then u is a leaf.

(i) $\Rightarrow$ (ii): Since $\ell[v_1]r[v_1]\ldots\ell[v_s]r[v_s]$ is a subsequence of the frontier of u, if the frontier of u satisfies every color, then certainly so does $\ell[v_1]r[v_1]\ldots\ell[v_s]r[v_s]$.

(ii) $\Rightarrow$ (i): Suppose (ii) holds, and let the frontier of u be $w_1\ldots w_t$. Suppose for a contradiction that (i) does not hold. A counterexample to (i) would be a consecutive subsequence $w_i\ldots w_j\ldots w_k$ of the frontier of u such that $cl(w_i) = cl(w_k) \neq cl(w_j)$. Choose such a counterexample of smallest length $j - i + 1$. If $w_i\ldots w_k$ is contained within the frontier of some child of u, that frontier does not satisfy $cl(w_i)$, so apply the inductive hypothesis to obtain a contradiction. Thus w_i and w_k must belong to the frontiers of distinct children $u_{\hat{i}}$ and $u_{\hat{k}}$ of U ($\hat{i} < \hat{k}$). Hence the color $c = cl(w_i)$ is incomplete at $u_{\hat{i}}$ and $u_{\hat{k}}$, so, by (2), $c \in \{cl(\ell[u_{\hat{i}}]), cl(r[u_{\hat{i}}])\}$ and $c \in \{cl(\ell[u_{\hat{k}}]), cl(r[u_{\hat{k}}])\}$. By (ii), since $\hat{i} < \hat{k}$, $cl(r[u_{\hat{i}}]) = cl(\ell[u_{\hat{k}}]) = c$.

Let $u_{\hat{j}}$ be the child of u to whose frontier w_j belongs, so $\hat{i} \leq \hat{j} \leq \hat{k}$. Since $\hat{i} < \hat{k}$, either $\hat{i} < \hat{j}$ or $\hat{j} < \hat{k}$. In the former case, (ii) implies that $cl(L[u_{\hat{j}}]) = c$, but then $\ell[u_{\hat{j}}]\ldots w_j$ is a counterexample to (i) of length smaller than that of $w_i\ldots w_k$, contradicting the choice of $w_i\ldots w_k$. The case $\hat{j} < \hat{k}$ is analogous. □

Lemma 4.6 *Assume (2) holds of T'. Then (ii) of Lemma 4.5 is equivalent to*

(iii) *For each descendent v of u, for each color c incomplete at a child of v, the children with color c form a consecutive subsequence of the children of v, and any uncovered child v_j with color c is an endpoint of this subsequence—either $cl(r[v_j]) = c$ and v_j is a left endpoint, or $cl(\ell[v_j]) = c$ and v_j is a right endpoint.*

proof Note first that if c is incomplete at a child v_i of v, then $c \in \{cl(\ell[v_i]), cl(r[v_i])\}$ by (2). To show that (iii) implies (ii), we merely note in addition that if v_i is covered by c, then certainly $cl(\ell[v_i]) = cl(r[v_i]) = c$. To show that (ii) implies (iii), suppose v_j is uncovered. Assume (ii), so by Lemma 4.5, the frontier of v_j satisfies c. But then, since v_j is not covered by c, only one of $cl(\ell[v_j])$ and $cl(r[v_j])$ can be c. Then (iii) follows from (ii). □

With Lemmas 4.5 and 4.6 in hand, we proceed to describe the second stage. This stage consists of two phases, phase A and phase B. In phase A, each node v ensures that the ordering of its children is consistent with the coloring (possibly by adding new nodes), and assigns colors to temporary storage locations (called *fields*) associated with itself and its children. In phase B, the values of the fields are processed, and the order of children of some nodes is reversed. Then some nodes are renamed to be R-nodes.

We let T^* denote the result of executing phase A on T, and let $\hat{T}$ denote the result of executing phase B on T^*.

Phase A is outlined in Figures 9 and 10.

Remarks:

1. For the purposes of this algorithm we associate with each node v four fields: *v.left*, *v.right*, *v.request-left* and *v.request-right*. We assume that these are all cleared initially. The values may be discarded after the completion of phase B.

2. The node v assigns a color c to its field *v.left* to signify that the leftmost element of v's frontier is colored c. Similarly for the field *v.right*. The parent of v assigns a color c to the field *v.request-left* to signify that the leftmost element of v's frontier "should be" colored c (from the parent's perspective). Similarly for *v.request-right*. If any of these fields is set, its value is some color incomplete at v. Moreover, each color incomplete at v is either *v.left* or *v.right* and either *v.request-left* or *v.request-right*. Fields of v are set only if v is uncovered.

3. If v is a P-node (Figure 10), new nodes are inserted between v and its original children. Figure 11 illustrates the new descendents of v introduced in phase A. A new node may be created which ends up having fewer than two children. Such a situation may be easily remedied during the post-processing phase: if a new node has only one child, identify it with its child, and if a new node has no children, delete it.

4. Fields are associated with only original nodes, never with newly created nodes.

5. At the end of phase A, every original node has become a Q-node.

In phase B, we use the following notation. If $\{a, b\} = \{a', b'\}$, we write $swapped(\langle a, b\rangle, \langle a', b'\rangle)$ for the predicate that is *true* if $a = b'$ and $b = a'$, and *false* if $a = a'$ and $b = b'$. We write $\oplus$ to denote "exclusive-or" ($GF(2)$ addition). Note that

$$swapped(\langle a, b\rangle, \langle a'', b''\rangle) = swapped(\langle a, b\rangle, \langle a', b'\rangle) \bigoplus swapped(\langle a', b'\rangle, \langle a'', b''\rangle) \quad (3)$$

Note also that in phase B we use two more temporary fields for each original node v, namely *v.opposite-parent* and *v.reverse*.

Note that step B3 of phase B can be done using standard parallel pointer-hopping techniques (or parallel tree expansion).

For a Q-node or R-node v in PQ-trees T_1 and T_2, define $flipped_v(T_1, T_2)$ to be *true* if the order of children of v in T_1 is the reverse of the order in T_2, *false* if the order is the same, and undefined otherwise. Note that

$$flipped_v(T_1, T_3) = flipped_v(T_1, T_2) \bigoplus flipped_v(T_2, T_3) \quad (4)$$

In the following, we will use T for the original PQ-tree, T^* for the result of applying phase A to T, $\hat{T}$ for the result of applying phase B to T^*, and $\hat{T}'$ for a PQ-tree equivalent

Q1 For each Q-node v with children $v_1 \ldots v_s$,

Q2 for each color c incomplete at a child of v,

Q3 check that the children with color c form a consecutive subsequence $v_i \ldots v_j$. (If not, halt.)

Q4 Check that any uncovered child with color c is either v_i or v_j

Q5 —if v_i, then assign $v_i.\textit{request-right} := c$,

Q6 —if v_j, then assign $v_j.\textit{request-left} := c$

Q7 (If neither, halt.)

Q8 Assign to each color $c \in \text{INCOMPLETE}(v)$ a distinct representative v_c drawn from $\{v_1, v_s\}$ such that $c \in \text{INCOMPLETE}(v_c)$.

Q9 (If this step is not possible, *halt.*)

Q10 For each color $c \in \text{INCOMPLETE}(v)$,

Q11 If v_c is not covered, then

Q12 If $v_c = v_1$, assign $v_c.\textit{request-left} := c$ and assign $v.\textit{left} := c$.

Q13 If $v_c = v_s$, assign $v_c.\textit{request-right} := c$ and assign $v.\textit{right} := c$.

Figure 9: Phase A for Q-nodes

to $\hat{T}$. See Figure 13. If v is a node of $\hat{T}'$ other than the root, we denote its parent in $\hat{T}'$ by $p(v)$.

Lemma 4.7 *For any PQ-tree $\hat{T}' \cong \hat{T}$ and R-node v of T,*
$\textit{flipped}_v(T^*, \hat{T}') \oplus \textit{swapped}(\langle v.\textit{left}, v.\textit{right}\rangle, \langle v.\textit{request-left}, v.\textit{request-right}\rangle) = \textit{flipped}_{p(v)}(T^*, \hat{T}')$

proof We first prove the claim for $\hat{T}' = \hat{T}$. In step B2 of phase B, we set

$$v.\textit{opposite-parent} := \textit{swapped}(\langle v.\textit{left}, v.\textit{right}\rangle, \langle v.\textit{request-left}, v.\textit{request-right}\rangle).$$

In step B2, we have

$$\begin{aligned} v.\textit{reverse} &:= \bigoplus_{u \text{ an ancestor of } v} u.\textit{opposite-parent} \\ &= v.\textit{opposite-parent} \oplus \left(\bigoplus_{u \text{ an ancestor of } p(v)} u.\textit{opposite-parent} \right) \\ &= v.\textit{opposite-parent} \oplus p(v).\textit{reverse} \end{aligned}$$

and by step B3, $\textit{flipped}_v(T^*, \hat{T}) = v.\textit{reverse}$ and $\textit{flipped}_{p(v)}(T^*, \hat{T}) = p(v).\textit{reverse}$. So the lemma follows for $\hat{T}$.

Now for any $\hat{T}' \cong \hat{T}$, $\textit{flipped}_v(\hat{T}, \hat{T}') = \textit{flipped}_{p(v)}(\hat{T}, \hat{T}')$ because v is an R-node in T. The lemma follows for $\hat{T}'$ by (4). □

P1 For each P-node v,

P2 create a new P-node w_c for each color c incomplete at a child of v.

P3 Construct a graph G_v by connecting each w_c to all the uncovered children of v with color c. If any w_c has > 2 neighbors, halt. Otherwise, G_v has maximum degree ≤ 2. Using known pointer-hopping techniques, identify the connected components of G_v, and verify that each is a simple path—not a cycle (otherwise, halt). Call these paths *color chains.*

P4 For each color chain χ,

P5 choose one of the two orientations of χ arbitrarily.
Construct a new Q-node u_χ and assign the nodes of χ as children of u_χ in the chosen order.

P6 For each node w_c, let the children of w_c be those children of v covered by c.

P7 If there is only one color chain χ, and every original child of v is now a child or grandchild of u_χ, identify v with u_χ, so that v is now a Q-node, and follow the algorithm of Figure 9. Otherwise continue with step P8.

P8 Construct a new P-node $\hat{v}$ whose children are
$\{u_\chi$: the chain χ does not contain any w_c, where $c \in$ INCOMPLETE$(v)\}$
$\cup \{y : y$ is a child of v at which no color is incomplete$\}$.

P9 Rename v to be a Q-node whose children are $\{\hat{v}\} \cup \{u_\chi$: the chain χ contains some w_c such that $c \in$ INCOMPLETE$(v)\}$,
ordered so that each such u_χ is a leftmost or rightmost child.

P10 For each color $c \in$ INCOMPLETE(v),

P11 check that w_c is either the leftmost or the rightmost child of its parent (else, halt). If in fact w_c is the *only* child of its parent, do nothing. Otherwise ...

P12 If w_c is a child of v's leftmost child,

P13 assign $v.left := c$.

P14 If w_c is the rightmost child of its parent, flip its parent (reverse the order of the parent's children).

P15 If w_c is a child of v's rightmost child,

P16 assign $v.right := c$.

P17 If w_c is the leftmost child of its parent, flip its parent.

P18 rename w_c's parent to be an R-node.

P19 For each color chain χ,

P20 For each uncovered child x appearing in χ

P21 if w_c is to the left of x, then assign $x.request\text{-}left := c$,

P22 and if w_c is to the right of x, then assign $x.request\text{-}right := c$.

Figure 10: Phase A for P-nodes

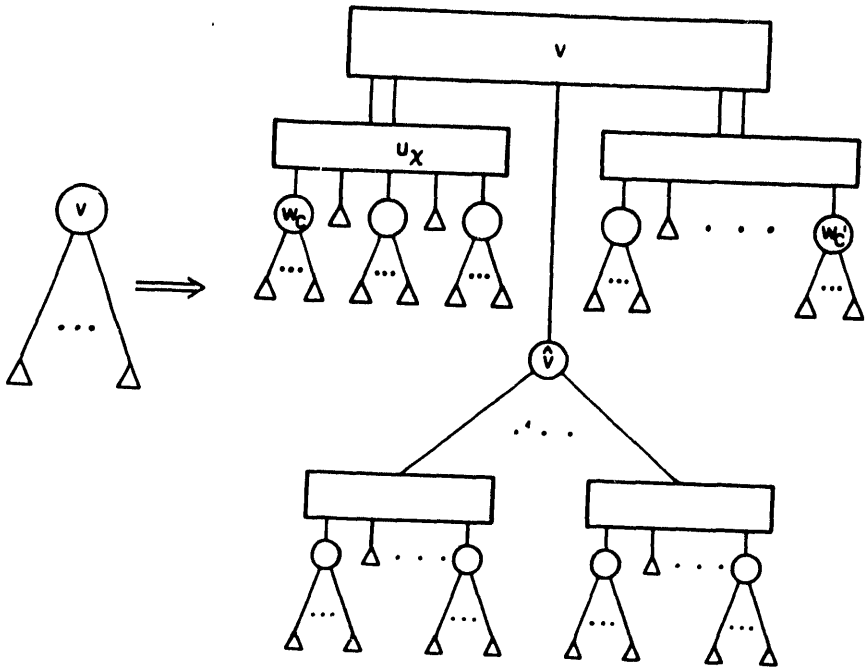

Figure 11: modifications to a P-node

B1 For each original node v other than the root,

B2

$$v.\textit{opposite-parent} := \textit{swapped}(\langle v.\textit{left}, v.\textit{right}\rangle, \langle v.\textit{request-left}, v.\textit{request-right}\rangle)$$

B3 For each original node v other than the root,

$$v.\textit{reverse} := \bigoplus_{u \text{ an ancestor of } v} u.\textit{opposite-parent}$$

B4 For each original node v other than the root,

B5 if $v.\textit{reverse}$ then flip v (reverse the order of its children).

B6 For each original node v other than the root,

B7 if v is not covered and INCOMPLETE(v) $\neq \emptyset$ then rename v to be an R-node.

Figure 12: Phase B

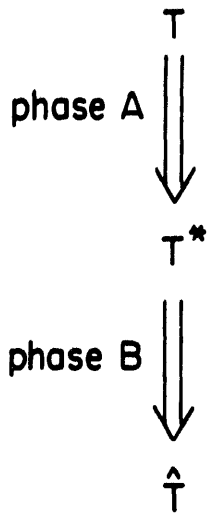

Figure 13: Phase A takes T to T^*, and phase B takes T^* to $\hat{T}$.

Lemma 4.8 *For any $\hat{T}' \cong \hat{T}$, for any node v of T, $INCOMPLETE(v) \subseteq \{cl(\ell_{\hat{T}'}[v]), cl(r_{\hat{T}'}[v])\}$. In particular, if v.request-left was assigned c, then*

$$c = \begin{cases} cl(\ell_{\hat{T}'}[v]) & \text{if not } flipped_{p(v)}(T^*, \hat{T}') \\ cl(r_{\hat{T}'}[v]) & \text{if } flipped_{p(v)}(T^*, \hat{T}') \end{cases}$$

and if v.request-right was assigned c, then

$$c = \begin{cases} cl(r_{\hat{T}'}[v]) & \text{if not } flipped_{p(v)}(T^*, \hat{T}') \\ cl(\ell_{\hat{T}'}[v]) & \text{if } flipped_{p(v)}(T^*, \hat{T}') \end{cases}$$

proof by induction on height of v. Trivial for any covered v, including v a leaf. Also trivial for INCOMPLETE$(v) = \emptyset$, including $v =$ the root. Therefore, say v is an uncovered node with parent $p(v)$ and $c \in$ INCOMPLETE(v), and suppose the lemma holds for the children of v. By our assumptions on v, either *v.request-left* or *v.request-right* was assigned c in phase A. Assume without loss of generality that *v.request-left* $= c$.

case a) v is a Q-node with children $v_1 \ldots v_s$ in T^*. In step Q8, $c \in$ INCOMPLETE(v_c), where $v_c \in \{v_1, v_s\}$. We treat the case in which $v_c = v_s$; the case $v_c = v_1$ is similar. If $v_c = v_s$, then *v_s.request-right* $:= c$ and *v.right* $:= c$ in step Q13. By the inductive hypothesis,

$$c = \begin{cases} cl(r_{\hat{T}'}[v_s]) & \text{if not } flipped_v(T^*, \hat{T}') \\ cl(\ell_{\hat{T}'}[v_s]) & \text{if } flipped_v(T^*, \hat{T}') \end{cases}$$

But the right-hand side is

$$= \begin{cases} cl(r_{\hat{T}'}[v]) & \text{if not } flipped_v(T^*, \hat{T}') \\ cl(\ell_{\hat{T}'}[v]) & \text{if } flipped_v(T^*, \hat{T}') \end{cases}$$

Moreover, by Lemma 4.7,

$$flipped_{p(v)}(T^*, \hat{T}') = flipped_v(T^*, \hat{T}') \bigoplus swapped(\langle v.left, v.right\rangle, \langle v.request\text{-}left, v.request\text{-}right\rangle)$$

Since $v.left = v.request\text{-}right$, $swapped(\cdot)$ is true, so $flipped_{p(v)}(T^*, \hat{T}') = \text{not } flipped_v(T^*, \hat{T}')$. The lemma follows.

case b) v is a P-node. In step P9, c is incomplete at either v's leftmost or rightmost child. Say c is incomplete at v's rightmost child (the other case is similar). This child is v_χ, where the color chain χ contains w_c. By step P17, w_c is the rightmost child of its parent u_χ in T^*. It remains to show

$$c = \begin{cases} cl(r_{\hat{T}'}[v]) & \text{if not } flipped_v(T^*, \hat{T}') \\ cl(\ell_{\hat{T}'}[v]) & \text{if } flipped_v(T^*, \hat{T}') \end{cases}$$

for then the lemma follows by applying Lemma 4.7 as in case (a).

But $flipped_v(T^*, \hat{T}')$ iff $flipped_{u_\chi}(T^*, \hat{T}')$ because u_χ was made an R-node in step P18. Hence if we let z be the rightmost node of u_χ in T^*, we have $r_{\hat{T}'}[v] = r_{\hat{T}'}[z]$ if not $flipped_v(T^*, \hat{T}')$ and $\ell_{\hat{T}'}[v] = \ell_{\hat{T}'}[z]$ if $flipped_v(T^*, \hat{T}')$. Formally w_c is the rightmost child of u_χ. Any child of w_c is covered by c, so if w_c has any children, the lemma follows. If w_c has no children, it will be deleted during the post-processing phase (see comment 3 on page 23). Hence in this case the true rightmost child of u_χ in T^* is an uncovered node y with color c appearing to the left of w_c in χ. In step P22, $y.request\text{-}right$ was assigned c, so by the inductive hypothesis,

$$c = \begin{cases} cl(r_{\hat{T}'}[y]) & \text{if not } flipped_{u_\chi}(T^*, \hat{T}') \\ cl(\ell_{\hat{T}'}[y]) & \text{if } flipped_{u_\chi}(T^*, \hat{T}') \end{cases}$$

The lemma then follows by remarks above. □

Lemma 4.9 *If there is a PQ-tree equivalent to T satisfying every color, then phases A and B succeed. Say phases A and B produce the PQ-tree $\hat{T}$. Then the orderings generated by $\hat{T}$ are exactly the orderings generated by T and satisfying every color.*

proof We show that phases A and B induce a frontier-preserving bijection ϕ from $\{T' | T' \cong T, T' \text{ satisfies every color}\}$ to $\{\hat{T}' | \hat{T}' \cong \hat{T}\}$. We first define ϕ, noting that it is frontier-preserving. We show that if there is a $T' \cong T$ satisfying every color, then phases A and B succeed with input T, producing a PQ-tree $\hat{T}$. Next we show that $\phi(T') \cong \hat{T}$. Finally, we show that for any $\hat{T}' \cong \hat{T}$, the inverse image $\phi^{-1}(\hat{T}')$ is a PQ-tree $T' \cong T$ and satisfying every color.

Let T' be a PQ-tree equivalent to T and satisfying every color. Lemma 4.2, (2) holds of T'. By Lemmas 4.5 and 4.6, condition (iii) holds of T'. It is a straightforward application of (2) and (iii) to the algorithms of Figures 9 and 10 to show that phase A can be successfully applied to T' to yield a PQ-tree $(T')^*$ such that each node v of T' has the same frontier in $(T')^*$ as in T'. Let $\widehat{T'}$ be the result of executing phase B on $(T')^*$. See Figure 14. An application of Lemma 4.8 to $(T')^*$ and $\widehat{T'}$ will show that for every

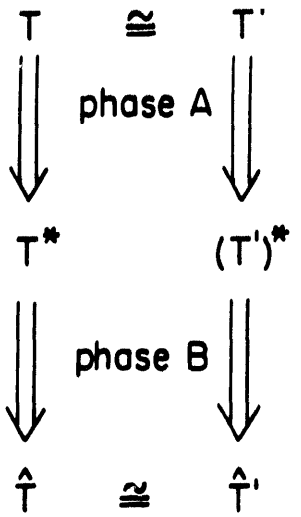

Figure 14: Phase A takes T to T^* and T' to $(T')^*$. Phase B takes T^* to $\hat{T}$, and $(T')^*$ to $\widehat{T'}$.

node v of T', $flipped_v((T')^*, \widehat{T'})$ is false, and hence $\widehat{T'}$ has the same frontier as $(T')^*$. We let $\phi(T') = \widehat{T'}$.

Now, we use the fact that Phase A succeeded for input PQ-tree T' to show that it succeeds for PQ-tree T (although in the latter case Phase A will generally involve reordering children). For any node v of T, if v is a Q-node, it has the same children in the same order (up to a possible flip) in T as in T'. If v is a P-node, node that the graph G_v constructed in step P2 depends only on the coloring of v and its children, which coloring is the same in T as in T'. Hence phase A succeeds for input PQ-tree T, producing a PQ-tree T^*. Let $\hat{T}$ be the result of executing phase B on T^*.

By carrying the argument a bit further, we see that $\hat{T}$ and $\widehat{T'}$ are equivalent. Let v be any node of T. If v is a Q-node, then certainly v has the same children in the same order (up to a flip) in $\hat{T}$ and $\widehat{T'}$. If v is a P-node, since the color chains are determined only by the coloring of v and its children, the nodes u_χ created in phase A have the same children in the same order (up to a flip) in $\hat{T}$ and $\widehat{T'}$. It follows that T^* and $(T')^*$ are equivalent. Hence $\hat{T}$ and $\widehat{T'}$ have the same nodes, each node v has the same children, and in the case that v is a Q-node, the order of v's children is the same (up to a flip). It remains to show that the nodes made into R-nodes in step B are consistent, i.e. if v is an uncovered node with some color incomplete, then $flipped_v(\hat{T}, \widehat{T'}) = flipped_{p(v)}(\hat{T}, \widehat{T'})$. But note that

$$
\begin{aligned}
&flipped_v(\hat{T}, \widehat{T'}) \bigoplus flipped_{p(v)}(\hat{T}, \widehat{T'}) \\
&\quad = flipped_v(\hat{T}, T^*) \bigoplus flipped_v(T^*, (T')^*) \bigoplus flipped_v((T')^*, \widehat{T'}) \\
&\qquad flipped_{p(v)}(\hat{T}, T^*) \bigoplus flipped_{p(v)}(T^*, (T')^*) \bigoplus flipped_{p(v)}((T')^*, \widehat{T'}) \\
&\qquad \text{by equation 4} \\
&\quad = [flipped_v(\hat{T}, T^*) \bigoplus flipped_{p(v)}(\hat{T}, T^*)] \bigoplus flipped_v(T^*, (T')^*) \bigoplus flipped_{p(v)}(T^*, (T')^*)
\end{aligned}
$$

because $flipped_v((T')^*, \widehat{T'}) = flipped_{p(v)}((T')^*, \widehat{T'}) = false$

$= [v.opposite\text{-}parent] \bigoplus flipped_v(T^*, (T')^*) \bigoplus flipped_{p(v)}(T^*, (T')^*)$

by line B5 of phase B.

By line B2 of phase B, $v.opposite\text{-}parent = swapped(\langle v.left, v.right\rangle, \langle v.request\text{-}left, v.request\text{-}righ$
Let $v.left$, $v.request\text{-}left$, etc. be the values assigned to these fields during the execution of phase A on T, and let $v.left'$, $v.request\text{-}left'$, etc. be the corresponding values assigned during the execution of phase A on T'. Then by phase A (steps Q6,Q5 and P22,P21), we can also substitute for the second and third term, obtaining

$$
\begin{aligned}
&= [swapped(\langle v.left, v.right\rangle, \langle v.request\text{-}left, v.request\text{-}right\rangle)] \\
&\quad \bigoplus swapped(\langle v.left, v.right\rangle, \langle v.left', v.right'\rangle) \\
&\quad \bigoplus swapped(\langle v.request\text{-}left, v.request\text{-}right\rangle, \langle v.request\text{-}left', v.request\text{-}right'\rangle) \\
&= false
\end{aligned}
$$

by applying equation 3, using the fact that $swapped(\langle v.left', v.right'\rangle, \langle v.request\text{-}left', v.request\text{-}rig$ $false$.

All that remains is to show that if $\hat{T}'$ is any PQ-tree equivalent to $\hat{T}$, then $\hat{T}' = \phi(T')$ for some unique PQ-tree T' equivalent to T and satisfying every color. For every node v of T, v appears in $\hat{T}'$ and a child of v in T is separated from v in $\hat{T}'$ by zero to three *new* nodes (i.e. nodes that do not appear in T). To obtain T' from $\hat{T}'$, eliminate these new nodes, connecting the original children of v directly to v *without reordering them.* Then rename each node to be either a P-node or a Q-node, depending on what it was in T. Let the result be T'. It is easy to verify that T' is equivalent to T: each node v has the same children, and if v is a Q-node, they are in the same order (up to a flip). Because we have not reordered children, this mapping $\phi^{-1}(\cdot)$ is frontier-preserving. Any two distinct trees equivalent to $\hat{T}$ have distinct frontiers, so map to distinct trees; thus T' is unique.

It remains to show that the frontier of T' satisfies every color. By Lemma 4.8, every color incomplete at an original node is an endpoint of the frontier of that node in $\hat{T}'$, and hence in T'. That is, (2) holds for T'. By Lemmas 4.5 and 4.6, to show that T' satisfies every color, it is sufficient to show that each node u of T' satisfies condition (iii).

let v be any node of T', and let c be a color incomplete at a child of v.

case a) v is a Q-node. By step Q3, the children of v with color c form a consecutive subsequence $v_i \ldots v_j$ of the children of v in T^*, and hence are also consecutive in $\hat{T}$, $\hat{T}'$, and finally T'. By step Q4, any uncovered child with color c is either v_i or v_j in T^*. Suppose without loss of generality it is v_i. Then $v_i.request\text{-}right$ was assigned c

in step Q5. By Lemma 4.8,

$$c = \begin{cases} cl(r_{\hat{T}'}[v_i]) & \text{if not } \mathit{flipped}_v(T^*, \hat{T}') \\ cl(\ell_{\hat{T}'}[v_i]) & \text{if } \mathit{flipped}_v(T^*, \hat{T}') \end{cases}$$

If not $\mathit{flipped}_v(T^*, \hat{T}')$ then v_i is still the left endpoint of the c-colored sequence. In this case, $cl(r_{T'}[v_i]) = cl(r_{\hat{T}'}[v_i]) = c$, and condition (iii) is satisfied. Similarly if $\mathit{flipped}_v(T^*, \hat{T}')$.

case b) v is a P-node. We first must show that the children of v with color c form a consecutive subsequence in T', and any uncovered child with color c is an endpoint of this subsequence. The following condition is equivalent: the *covered* children with color c form a consecutive subsequence (possibly empty), and each uncovered child with color c is either immediately to the left or immediately to the right of this subsequence. Now, the covered children with color c are all the children of w_c in $\hat{T}'$, hence are consecutive in $\hat{T}'$ and in T'. Any uncovered child y of v with color c is adjacent to w_c in the chain χ containing w_c, hence y is immediately to the left or right of w_c as a child of u_χ in $\hat{T}$. It follows that in T' the child y is adjacent to the sequence of covered children with color c. To conclude that condition (iii) holds, we must verify if y is to the left of the sequence in T', then $cl(\ell[y]) = c$, and if y is to the right of the sequence, then $cl(r[y]) = c$. This can be done as in case a. □

This completes the description of the disjoint-reduction algorithm. Lemma 4.9 proves the correctness of phases A and B, so we have proved Theorem 4.1.

4.3 Segregation and Joining

In this subsection, we make some observations and introduce some terminology useful to the rest of the algorithms in this section. We also describe the process of "joining" PQ-trees. If a PQ-tree T is reduced with respect to a set A of ground elements, the part of the tree "pertinent" to the set A is "contiguous," in a sense described below.

Lemma 4.10 *Let T be a PQ-tree reduced with respect to a set A with more than one element. Suppose $v = lca(A)$ has children $v_1 \ldots v_s$. For some consecutive subsequence of children $v_p \ldots v_q$ $(p < q)$, $A = \bigcup_{p \le i \le q} leaves(v_i)$.*

proof If only one child v_i of v satisfies the condition

$$leaves(v_i) \cap A \neq \emptyset \tag{5}$$

then $lca(A)$ is a descendant of v_i, contrary to choice of v. Thus v has more than one child v_i satisfying condition (5). Let the leftmost such child of v be v_p and the rightmost

v_q. Suppose that for some child v_i $(p \leq i \leq q)$, we did not have $leaves(v_i) \subset A$. Then either the frontier of T does not satisfy the subset A, or the frontier of the tree obtained from T by flipping v_i does not satisfy A. (or both). Hence we have $leaves(v_i) \subset A$ for each $p \leq i \leq q$. The claim follows because v was chosen to be an ancestor of every leaf in A. □

We say that T has been *segregated* with respect to A if $A = leaves(lca(A))$. By Lemma 4.10, a PQ-tree T reduced with respect to A can always be segregated with respect to A, possibly by introducing an R-node. For if $leaves(lca(A))$ contains ground elements not in A (i.e. if $p > 1$ or $q < s$ in Lemma 4.10), we can introduce into T an R-node v' as a child of $v = lca(A)$ replacing v's children $v_p \ldots v_q$. In this modified version of T, $v' = lca(A)$ and $A = leaves(v')$. The modification can be carried out within the same time bounds as reduction, for a suitable representation of PQ-trees.

If T is segregated with respect to A, we call the subtree rooted at $lca(A)$ the A-pertinent subtree of T, and denote it by $T|A$. Let $T|\overline{A}$ denote the result of deleting the nodes of $T|A$ from T.

Suppose that T is segregated with respect to A. If $lca_T(A)$ is an R-node, we say T is A-rigid; otherwise (if $lca(A)$ is a P-node or Q-node), we say that T is A-hinged. These definitions are motivated by the following considerations. If T is A-hinged, any equivalence transformations may be applied to any node of the A-pertinent subtree of T without affecting the rest of the tree. Consequently, we have the following lemma:

Lemma 4.11 *If T is A-hinged, and $\lambda, \tau \in L(T)$, then there is a $\gamma \in L(T)$ such that $\gamma|\overline{A} = \lambda|\overline{A}$ and $\gamma|A = \tau|A$.*

Now suppose T is A-rigid. Any equivalence transformation may be applied to any node of $T|A$ other than $lca(A)$ without affecting the rest of T. However, $lca(A)$ cannot be flipped without flipping its parent, because $lca(A)$ is an R-node. We obtain the following lemma:

Lemma 4.12 *If T is A-rigid, and $\lambda, \tau \in L(T)$, then either*

- *there is a $\gamma \in L(T)$ such that $\lambda|\overline{A} = \tau|\overline{A}$ and $\lambda|A = \gamma|A$,*

  or

- *there is a $\gamma \in L(T)$ such that $\lambda|\overline{A} = \tau|\overline{A}$ and $\lambda|A = (\gamma|A)^R$,*

  but not both.

Thus when T is A-hinged, the choice of an ordering on A is totally independent from the choice of an ordering on the rest of T's ground set, but when T is A-rigid, the choice of an ordering on A is independent only up to a flip.

We make use of the new terminology in showing how to compute the join of two PQ-trees.

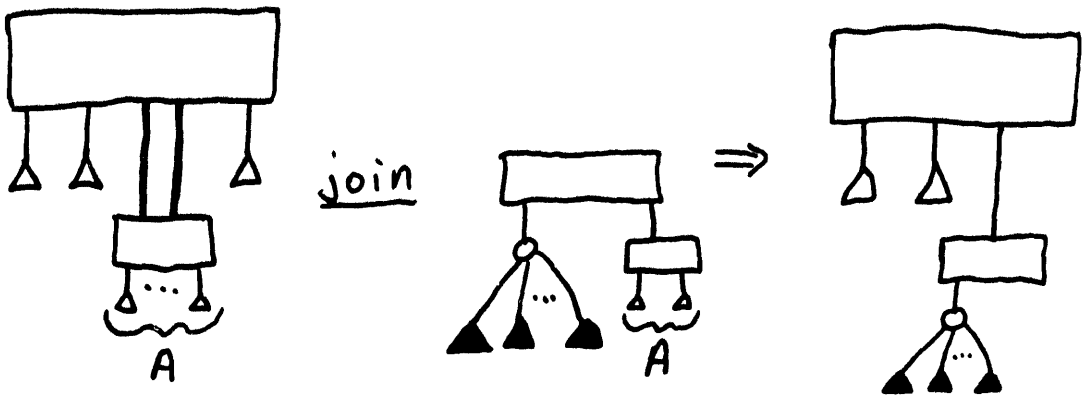

Figure 15: Computing the join of T_1 with T_2 in the case T_2 is A-hinged.

Recall the definition of *joining*, given in Subsection 4.1. Given two PQ-trees T_1 and T_2 over the respective ground sets S_1 and S_2, let $A = S_1 \cap S_2$, and suppose T_1 and T_2 are reduced and segregated with respect to A. For $\lambda_1 \in L(T_1)$ and $\lambda_2 \in T_2$, let λ_1 *join* λ_2 denote the ordering over $S_1 \cup S_2 - A$ obtained by substituting $\lambda_2|\overline{A}$ for $\lambda_1|A$ in λ_1. Let the join of T_1 with T_2 be the PQ-tree T generating

$$\{\lambda_1 \textit{ join } \lambda_2 \mid \lambda_1 \in L(T_1), \lambda_2 \in L(T_2), \text{ and } \lambda_1|A = (\lambda_2|A)^R\} \tag{6}$$

For the set (6) to be non-empty, there must be some ordering τ of A "agreed upon" by an ordering generated by T_1 and an ordering generated by T_2, i.e. for some $\lambda_1 \in L(T_1)$ and $\lambda_2 \in L(T_2)$, $\lambda_1|A = \tau = (\lambda_2|A)^R$. To determine whether the set (6) is non-empty (and find some agreed-upon ordering τ of A, we may intersect $T_1|A$ with $T_2|A$ and choose τ to be the frontier of the result, if the result is not the null tree. The intersection algorithm takes $O(\log^2 n)$ time. We can, however, *assume* that (6) is non-empty, and compute a PQ-tree T that is the join of T_1 and T_2, provided our assumption is correct. This computation takes only $O(\log n)$ time. We can later check our assumption using intersection—if the assumption is verified, fine; otherwise, the true join is the null tree.

For now, we will assume that (6) is non-empty, i.e. that there is an agreed-upon ordering τ of A. Given this assumption (but not given τ itself), we consider the problem of computing the join of T_1 with T_2 when T_2 is reduced with respect to $S_2 - A$ as well as with respect to A. This latter condition simplifies the interaction in T_2 between $T_2|A$ and $T_2|\overline{A}$.

The reader should observe throughout the remainder of this subsection that there is no difficulty in extending our techniques to computing the join of T_1 with many trees $T_2^{(j)}$ simultaneously, as long as the trees $T_2^{(j)}$ all have disjoint ground sets. Assigning one processor to each node of each tree, we can compute this "multiple join" in $O(\log n)$ time, where n is the total number of nodes.

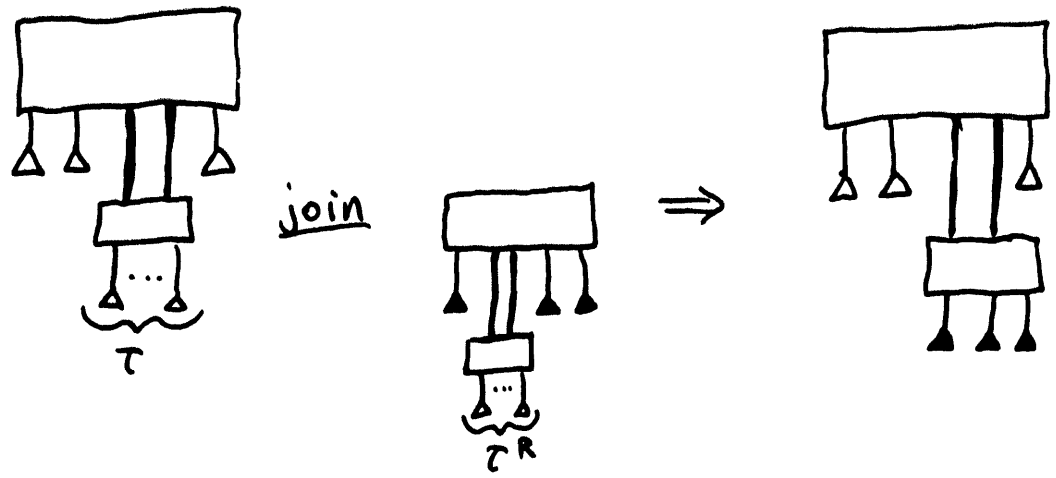

Figure 16: Computing the join of T_1 with T_2 in the case both T_1 and T_2 are A-rigid.

Segregate T_2 with respect to A and $S_2 - A$. By Lemma 4.10, in the segregated tree the root has two children, $lca_{T_2}(A)$ and $lca_{T_2}(S_2 - A)$.

There are two cases in computing the join T of T_1 and T_2. First, suppose that either T_1 or T_2 is A-hinged. Then let T be the PQ-tree obtained by substituting $T_2|\overline{A}$ for $T_1|A$ in T_1 (identifying the root of $T_2|\overline{A}$ with the root of $T_1|A$), as in Figure 15. We claim that T is the join of T_1 with T_2. For note that T generates

$$\{\lambda_1 \; join \; \lambda_2 \mid \lambda_1 \in L(T_1), \lambda_2 \in L(T_2)\} \tag{7}$$

so certainly $L(T)$ includes (6). But suppose $\lambda_1 \; join \; \lambda_2$ belongs to (7). By Lemma 4.11, there exist $\lambda_1' \in L(T_1)$ and $\lambda_2' \in L(T)$ such that $\lambda_i'|\overline{A} = \lambda_i|\overline{A}$ $(i = 1, 2)$ but $\lambda_1|A = \tau$ and $(\lambda_2|A)^R = \tau$. Thus $\lambda_1 \; join \; \lambda_2 = \lambda_1' \; join \; \lambda_2'$ is in (6), proving (6) and (7) are the same sets.

For the second case, suppose both T_1 and T_2 are A-rigid. This case is slightly more difficult. For simplicity, first suppose that the frontier of $lca_{T_1}(A)$ is either the same as the frontier of $lca_{T_2}(A)$, or its reverse. We may assume without loss of generality that it is the reverse, by flipping every node of T_2 if necessary. Now consider $T_2|\overline{A}$, the tree obtained from T_2 by deleting the nodes of $T_2|A$. The root r of $T_2|\overline{A}$ may have only one child (its second child in T_2 being $lca(A)$).

$$\text{Let } (T_2|\overline{A})' = \begin{cases} T_2|\overline{A} - r & \text{if } r \text{ has one child} \\ T_2|\overline{A} & \textit{otherwise} \end{cases}$$

Then we obtain the join T of T_1 and T_2 by substituting $(T_2|\overline{A})'$ for $T_1|A$, and renaming the root of $(T_2|\overline{A})'$ to be an R-node. See Figure 16.

Ordinarily we will not be so lucky as to find that the frontier of $lca_{T_1}(A)$ is the same as (or the reverse of) the frontier of $lca_{T_2}(A)$. There are two techniques to which we can turn. The first is to execute a single phase of INTERSECT, giving us enough information to carry out the join. This technique takes $O(\log n)$ time. The second technique is to

make use of other constraints on orderings to construct the join provisionally in constant time, and later call on the first technique to verify it (if the verification fails, the join is the null tree). We describe the first technique here; the second technique will come into play in the planarity algorithm, and we treat it in Subsection 4.5.

Let $v = lca_{T_1}(A)$, and suppose v has children $v_1, \ldots, v_s$ ($s > 1$). Let $A_i = leaves(v_i)$ for $i = 1, \ldots s$. Because T_1 is A-rigid, we know that v is an R-node. Hence any ordering λ_1 generated by T_1 satisfies the following three constraints:

(a) λ_1 satisfies each of $A_1, \ldots, A_s$.

(b) λ_1 satisfies each of $A_1 \cup A_2, A_3 \cup A_4, \ldots$

(c) λ_1 satisfies each of $A_2 \cup A_3, A_4 \cup A_5, \ldots$

Constraint (a) is satisfied simply because each of the sets lies below a different child of v. Constraints (b) and (c) are satisfied because of the order imposed on its children by the R-node v.

Certainly any ordering τ of A agreed upon by an ordering λ_1 generated by T_1 and an ordering λ_2 generated by T_2 must satisfy constraints (a), (b), and (c), because any λ_1 satisfies these constraints. To determine whether any λ_2 satisfies these contraints, we reduce T_2 three times, once for each constraint. If the result of the reductions is the null tree, there is no agreed-upon ordering τ of A, and hence the join is the null tree. Otherwise, if all the reductions succeed, we segregate T_2 with respect to $A_1, \ldots, A_s$, obtaining nodes $v_1', \ldots, v_s'$ in T_2 such that $v_i' = lca(A_i)$. Because of contraints (b) and (c), which now hold in T_2, the order of these nodes in T_2 must be either $v_1 \ldots v_s$ or $v_s \ldots v_1$. We may assume the latter without loss of generality, flipping every node of T_2 if necessary.

Assuming that there exist $\lambda_1 \in L(T_1)$ and $\lambda_2 \in L(T_2)$ such that $\lambda_1|A = (\lambda_2|A)^R$, these orderings may be obtained as frontiers of T_1 and T_2 respectively, following the application of equivalence transformations to the subtrees rooted at the nodes $v_1, \ldots, v_s$ and $v_s', \ldots, v_1'$ respectively. We need not actually carry out these transformations in order to form the join; we merely follow the procedure outlined above as if the frontier of $lca_{T_1}(A)$ were the reverse of the frontier of $lca_{T_2}(A)$. This completes our description of the join operation. Observations concerning *join* in a special case useful in planarity-testing may be found in Subsection 4.5.

4.4 Intersection

Recall from Subsection 4.1 the definition of intersection: A PQ-tree T is the *intersection* of two PQ-trees T_1 and T_2 over the same ground set if $L(T) = L(T_1) \cap L(T_2)$. In this subsection, we describe an algorithm INTERSECTION(T_1, T_2) for intersecting two PQ-

trees using disjoint reduction as a subroutine. The algorithm modifies T_1 to be the intersection of the two original trees. If n is the size of the ground set, the algorithm runs in $O(\log^2 n)$ time using n processors, as stated in Theorem 4.2.

proof of Theorem 4.2. We can carry out intersection by $O(\log n)$ invocations of DISJOINT-REDUCE. The idea is to consider each node v of T_2 as a constraint on permitted orderings, and to incorporate this constraint into T_1 by reduction.

Assign to each node v of T_2 a distinct color $c(v)$. Let v be any node of T_2 with children $v_1 \ldots v_s$. Let $A = leaves_{T_2}(v)$, and let $A_i = leaves_{T_2}(v_i)$ for $i = 1, \ldots, s$. We associate with v the following two constraints:

(i) the elements of A must be consecutive, and

(ii) the elements of A_i must be consecutive for each i.

Constraint (i) can be incorporated into T_1 by reducing T_1 with respect to A. Constrain (ii) can be incorporated into T_1 by reducing T_1 with respect to the coloring $cl(A_i) = c(v_i)$. If v is a Q-node, there is also an additional constraint associated with v:

(iii) the subsequence of elements of A_1 must be adjacent to the subsequence of elements of A_2, which must be adjacent to the subsequence of elements of A_3, etc.

This additional constraint can be incorporated into T_1 by reducing T_1 twice, once with respect to the coloring $cl(A_i) = c(v_{\lceil i/2 \rceil})$ and once with respect to the coloring $cl(A_i) = c(v_{\lfloor i/2 \rfloor})$. The first coloring forces A_1 to be adjacent to A_2, A_3 to be adjacent to A_4, etc. The second coloring forces A_2 to be adjacent to A_3, A_4 to be adjacent to A_5, etc. Note that in each of these colorings the only nodes to receive colors are those below v.

To intersect T_1 with T_2, we need only incorporate into T_1 the constraints associated with each node v of T_2. We could do this by sequentially carrying out at most four reductions of T_1 for each node of T_2; however, this would be inefficient, as T_2 may have $\Omega(n)$ nodes. Our goal, therefore, is to perform only $O(\log n)$ reductions, where necessarily each reduction will incorporate into T_1 the constraints associated with many nodes of T_2. To make this possible, we introduce a useful tool for manipulating trees.

Fix a tree T of n nodes. If $n \geq 2$, let a *good separator* of T be a node v such that if $T^{(1)}, \ldots, T^{(k)}$ are the subtrees of T rooted at the children of v, and $T^{(0)}$ is the tree obtained from T by deleting $T^{(1)}, \ldots, T^{(k)}$, then each subtree $T^{(0)}, \ldots, T^{(k)}$ has no more than $\frac{n}{2}$ nodes. It is easy to find a good separator if $n \geq 2$. For each node u of the tree T, let $size(u)$ be the number of nodes in the subtree rooted at u, and let $childSize(u)$ be the maximum number of nodes in the subtree rooted at a child of u. Note that $size$ and $childSize$ can be computed using parallel tree contraction. Let u be a node such that $size(u) > \frac{m}{2}$ but $childSize(u) \leq \frac{m}{2}$. Then u is a good separator. To see that such a u exists, consider a path $v_1 \ldots v_s$ from the root v_1 of T to a leaf v_s such that $size(v_{i+1}) = childSize(v_i)$. Since the first node v_1 has $size(v_1) = m > \frac{m}{2}$ and the last

node v_s has $size(v_s) = 1 \leq \frac{m}{2}$, some node along this path must satisfy the condition above. We have shown

Claim Given a tree T of m nodes, a good separator v of T may be found in $O(\log n)$ time using m processors.

We will now describe the algorithm for intersecting a PQ-tree T_1 with another PQ-tree T_2, proving Theorem 4.4. The form of the algorithm is a simple divide-and-conquer. If T_2 has only one node, do nothing. Otherwise, choose a separator v of T_2 with children $v_1 \ldots v_s$. Introduce into T_1 the constraints associated with v as described above. Namely, reduce T_1 with respect to each of the following colorings in turn:

(a) $cl(leaves_{T_2}(v)) = c$ (all other elements uncolored),

(b) $cl(leaves_{T_2}(v_i)) = c(v_i)$ for all $1 \leq i \leq s$,

and, if v is a Q-node,

(c) $cl(leaves_{T_2}(v_i)) = c(v_{\lceil i/2 \rceil})$ for all $1 \leq i \leq s$, and

(d) $cl(leaves_{T_2}) = c(v_{\lfloor i/2 \rfloor})$ for all $1 \leq i \leq s$.

Next, we use the separator to separate T_1 and T_2. Conceptually, we remove the edges from v to its children in T_2, so that T_2 falls into small pieces. Segregate T_1 with respect to the coloring (a), and let v' be the node of T_1 such that $leaves_{T_1}(v') = leaves_{T_2}(v)$. Similarly, segregate T_1 with respect to the coloring (b), and let v_i' be the node of T_1 such that $T_1(v_i') = leaves_{T_2}(v_i)$. Now conceptually remove the edges connecting v' to its children and the edges connecting each v_i' to its parent, so that T_1 falls into small pieces. For each piece of T_2, there is a corresponding piece of T_1 with the same leaves (where we identify v' in T_1 with v in T_2, and v_i' in T_1 with v_i in T_2. Now intersect the parts of T_1 with the corresponding parts in T_2 recursively.

This concludes our description of the intersection algorithm and the proof of Theorem 4.4. □

4.5 Representing Cyclic Orderings with PQ-trees

In our planarity algorithm, the orderings represented by PQ-trees will have a special form. The ground set of each PQ-tree T will be the disjoint union of two non-empty sets $in(T)$ and $out(T)$, and in every ordering in $L(T)$, the elements of each of these sets will be consecutive. That is, T will be reduced with respect to each of these sets. We will say for conciseness that T is (in,out)-reduced.

Moreover, we intend to represent not sets of *linear* orderings, but sets of *cyclic* orderings. Consider Figure 17. We have depicted one cyclic ordering, namely $(a_4a_3a_2a_1b_1b_2b_3)$. (This notation means that a_4 preceeds a_3 and follows b_3 in the cyclic order.) One way to represent this cyclic ordering as a linear ordering is $a_4a_3a_2a_1b_1b_2b_3$; another is

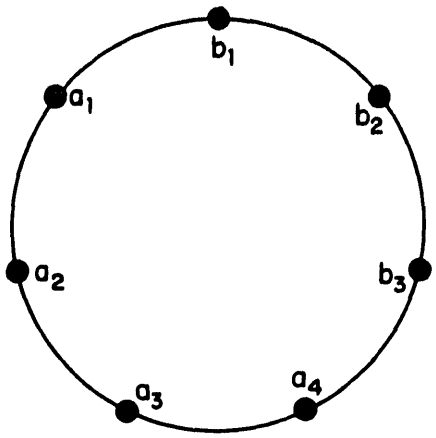

Figure 17: The cyclic ordering $(a_4a_3a_2a_1b_1b_2b_3)$.

$b_1b_2b_3a_4a_3a_2a_1$, and still another is $a_2a_1b_1b_2b_3a_4a_3$. Thus there is considerable redundancy in representing a cyclic ordering as a linear ordering. Note that in the third way, the elements of IN_T are not consecutive. We will rule out such representations as invalid, and restrict ourselves to representing a cyclic ordering as a linear ordering in which all the elements of $in(T)$ are consecutive and all the elements of $out(T)$ are consecutive. Consequently, there are two valid representations of a given cyclic order: one in which the elements of $in(T)$ come first, and another in which the elements of $out(T)$ come first. Say $\alpha\beta$ is one valid representation of a cyclic ordering, where α consists of elements of $in(T)$ and β consists of elements of $out(T)$. Then the other valid representation of the same cyclic ordering is $\beta\alpha$.

To represent a set of cyclic orderings, we can use a PQ-tree T that generates valid representations of these orderings. If T is (in,out)-reduced, let $C(T)$ be the set of cyclic orderings whose valid representations are generated by T. Other definitions carry over from linear orderings. A cyclic ordering σ of S is said to satisfy a subset $A \subset S$ if the elements of A are consecutive in σ. We let $\sigma|A$ denote the *linear* ordering of A induced by σ, and let $\sigma|\overline{A}$ denote the linear ordering of $S - A$ induced by σ.

PQ-trees are useful because they can be reduced; hence, it is important to verify that our reduction algorithm will work on a PQ-tree T used to represent a set of cyclic orderings. If we reduce T with respect to a set A which is wholly contained in either $in(T)$ or $out(T)$, then reduction will work as expected—the reduced tree will represent the subset of $C(T)$ in which the elements of A are consecutive. For we may assume that T is segregated with respect to $in(T)$ and $out(T)$. If, for example, A is wholly contained in $in(T)$, reducing T with respect to A affects only the subtree $T|in(T)$. But for any cyclic ordering σ, the induced ordering $\sigma|in(T)$ is a *linear* ordering. Hence the reduction

operates on $T|in(T)$ correctly. It follows that MULTIPLE-DISJOINT-REDUCE(T, cl) will work as expected if each color class of cl is wholly contained in either $in(T)$ or $out(T)$. The result will still be (in,out)-reduced.

We also want to use the join operation on cyclic orderings. The join of two orderings, defined in Subsection 4.1, may be interpreted to apply to cyclic orderings: Suppose T_1 and T_2 are (in,out)-reduced PQ-trees over S_1 and S_2, respectively, and are reduced with respect to $A = S_1 \cup S_2$. For $\sigma_1 \in C(T)$ and $\sigma_2 \in C(T)$, we let σ_1 *join* σ_2 denote the *cyclic* ordering obtained by substituting the *linear* ordering $\sigma_2|A$ for the linear ordering $\sigma_1|A$ in σ_1. The *(in,out)-reduced join* of T_1 with T_2 is a PQ-tree T such that

$$C(T) = \{\sigma_1 \textit{ join } \sigma_2 : \sigma_1 \in L(T_1), \sigma_2 \in L(T_2), \sigma_1|A = (\sigma_2|A)^R, \text{ and} \quad (8)$$

$$\sigma_1 \textit{ join } \sigma_2 \text{ satisfies } in(T) \text{ and } out(T)\} \quad (9)$$

where $in(T) = in(T_1) \cup in(T_2) - A$ and $out(T) = out(T_1) \cup out(T_2) - A$.

The remainder of this subsection is devoted to showing how to compute the (in,out)-reduced join in two special cases, the two cases arising in the planarity algorithm of Section 6. In each case, A will be wholly contained in each of the sets $in(T_1)$ and $out(T_2)$.

Special case 1 holds if $A = out(T_2)$. In this case, the elements of $\sigma_2|\overline{A}$ are all elements of $in(T_2)$. Thus in obtaining σ_1 *join* σ_2, we substitute a sequence of elements of $in(T_2)$ for a sequence of elements of $in(T_1)$. The result is certainly reduced with respect to $in(T) = in(T_1) - A \cup in(T_2)$. It is also reduced with respect to $out(T) = out(T_1)$ because σ_1 is. The join takes place entirely in $T_1|in(T_1)$. Thus we may perform the ordinary join of $T_1|in(T_1)$ with T_2 as described in Subsection 4.3, and the result will be automatically (in,out)-reduced. Note that, as mentioned in Subsection 4.3, we may simultaneously join T_1 with many PQ-trees $T_2^{(j)}$ having disjoint ground sets.

For special case 2, on the other hand, the requirement that the result be (in,out)-reduced is not automatically satisfied. For this case, we take pains to compute the (in,out)-reduced join without performing a reduction. In fact, we develop a special form such that an n-node PQ-tree may be put in this form in $O(\log n)$ time, and subsequently join to another PQ-tree in constant time (subject to later verification). We take advantage of the redundancy of representation of cyclic orderings in obtaining this special form. First, we give the special form in Lemma 4.13. Next, we state special case 2 and show how the (in,out)-reduced join can be computed in this case. Finally, we prove Lemma 4.13.

Lemma 4.13 *Let T be a PQ-tree such that $out(T)$ is the disjoint union of non-empty sets B and C, and T is reduced with respect to B and C, as well as $in(T)$ and $out(T)$. Then there is a PQ-tree T' such that $C(T') = C(T)$ in which the root is a Q-node with children a, b, c such that $in(T) = leaves(a)$, $B = leaves(b)$, and $C = leaves(c)$.*

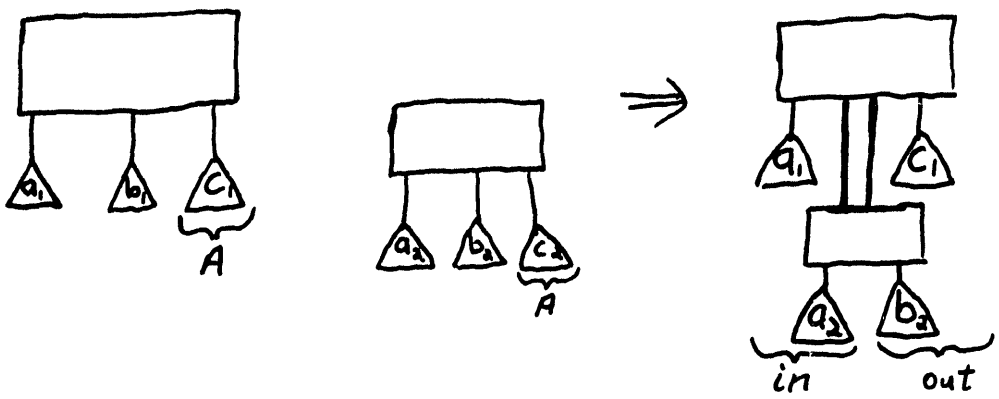

Figure 18: Replace the subtree rooted at the middle node of T_1 with $T_2|\overline{A}$, making the root of this new subtree an R-node.

Note: Lemma 4.13 also holds when "$in(T)$" and "$out(T)$" are exchanged.

We now state special case 2. It holds when $in(T_1) - A$ and $out(T_2) - A$ are both non-empty, T_1 is reduced with respect to $in(T_1) - A$, and T_2 is reduced with respect to $out(T_2) - A$.

To compute the (in,out)-reduced join of T_1 with T_2 in special case 2 (subject to later verification using intersection), first use Lemma 4.13 to modify the two PQ-trees. We modify T_1 so that the root r_1 of T_1 has three children $a_1b_1c_1$, where $leaves(a_1) = in(T_1) - A$, $leaves(b_1) = A$, and $leaves(c_1) = out(T_1)$. We modify T_2 so that the root r_2 of T_2 has three children $a_2b_2c_2$, where $leaves(a_2) = in(T_2)$, $leaves(b_2) = out(T_2) - A$, and $leaves(c_2) = A$. Carrying out these modifications of T_1 and T_2 is the *preprocessing stage.* To modify a tree with n leaves takes $O(\log n)$ time using n processors.

Verifying that (8) is non-empty can be done exactly as in Subsection 4.3, using constraints (a), (b), and (c) as well as intersection to check that some equivalence transformations can be applied to the nodes of $T_1|A$ and $T_2|A$ so that the frontier of one is the reverse of the frontier of the other.

For now, assume that (8) is non-empty. To compute the (in,out)-reduced join, replace the subtree of T_1 rooted at b_1 with $T_2|\overline{A}$, renaming r_2 (which replaced b_1 as a child of r_1) to be an R-node, as depicted in Figure 18. Let T be the resulting PQ-tree.

Before we show that T is in fact the (in,out)-reduced join of T_1 and T_2, note that computing T is very easy. In particular, after the preprocessing stage, the join can be done sequentially in constant time, assuming the pointer-based representation for PQ-trees recommended in Subsection 4.1. Of course, it is necessary to have a post-processing phase to eliminate the newly introduced R-node, but this can also be done in constant time because there is only one R-node, and it is a child of the root.

Now we will prove the correctness of the procedure given above for computing the

(in,out)-reduced join T of T_1 with T_2. To show that T is the (in,out)-reduced join, first suppose that σ_1 *join* σ_2 belongs to the set (8). Then $\sigma_1 \in C(T_1)$, $\sigma_2 \in C(T_2)$, $sigma_1|A = (\sigma_2|A)^R$, and σ_1 *join* σ_2 satisfies $in(T) = in(T_1) - A \cup in(T_2)$ and $out(T) = out(T_2) - A \cup out(T_1)$. For some $T_i' \cong T_i$, $fr(T_i')$ represents the cyclic ordering σ_i (for $i = 1, 2$). Let T' be the tree obtained by substituting $T_2'|A$ for $T_1'|A$ in T_1', renaming the root of $T_2'|A$ (now a child of the root of T_1') to be an R-node. It remains to show that $T' \cong T$.

We call on some notation defined in Subsection 4.2. For a Q-node or R-node v in PQ-trees $T^{(1)}$ and $T^{(2)}$, define $flipped_v(T^{(1)}, T^{(2)})$ to be *true* if the order of children of v in $T^{(1)}$ is the reverse of the order in $T^{(2)}$, *false* if the order is the same, and undefined otherwise.

Claim $flipped_{r_1}(T_1, T_1')$ *iff* $flipped_{r_2}(T_2, T_2')$

(Proof of the claim is given below).

We can now show that $T' \cong T$. Since $T_1' \cong T_2$, T_1 is the result of applying a number of equivalence transformations to T_1'. A subset of these equivalence transformations affect nodes of T_1' which are in T'. Apply this subset of transformations to the nodes of T_1 that are in T'. Similarly, consider the equivalence transformations used to transform T_2' to T_2, and apply to the nodes of T_2' in T' those transformations affecting those nodes. By the claim, if one of the first subset of transformations flips the root r_1 of T_1, one of the second flips the root r_2 of T_2. Thus we can legally carry out all these transformations on T' despite the fact that in T' the node r_2 is an R-node and a child of r_1. Let T'' be the result of applying all these transformations to T'. We claim that T'' is T. Indeed, the subtree of T'' rooted at r_2 is $T_2|\overline{A}$, and the subtree obtained from T'' by deleting every descendent of r_2 is $T_1|\overline{A}$.

In order to show that T is the (in,out)-reduced join, it remains only to show that every linear ordering λ generated by T represents a cyclic ordering belonging to the set (8), assuming the set is non-empty. First note that by the construction λ satisfies $in(T)$ and $out(T)$. Second, by the assumption that the set (8) is not empty, there is some linear ordering τ of A such that $lca_{T_1}(A)$ generates τ and $lca_{T_2}(A)$ generates $(\tau)^R$.

Assume without loss of generality that λ begins with an element of $in(T_1)$. By construction of T, λ can be written as $\alpha_1\alpha_2\beta_2\beta_1$, where $\alpha_i\beta_i = \lambda_i|\overline{A}$ for some ordering $\lambda_i \in L(T_i)$ (for $i = 1, 2$). By Lemma 4.12, there is a $\lambda_i' \in L(T_i)$ such that $\lambda_i'|\overline{A} = \lambda_i|\overline{A}$ and $\lambda_i'|A$ is either τ or $(\tau)^R$ (for $i = 1, 2$). Let σ_i' be the cyclic ordering represented by λ_i' (for $i = 1, 2$). If $\lambda_1'|A = \tau$ and $\lambda_2'|A = (\tau)^R$, or $\lambda_1'|A = (\tau)^R$ and $\lambda_2'|A = \tau$, then we are done, for then σ_1' *join* σ_2' belongs to the set (8). Assume without loss of generality that $\lambda_1'|A = \lambda_2'|A = \tau$. If T_2 is A-hinged, we can use Lemma 4.11 to obtain an ordering $\lambda_2'' \in L(T_2)$ such that $\lambda_2''|\overline{A} = \lambda_2|\overline{A}$, and proceed as above. Similarly if T_1 is A-hinged.

Hence assume that T_1 and T_2 are both A-rigid. It follows that for any two orderings $\lambda_1 \in L(T_1)$ and $\lambda_2 \in L(T_2)$, if $\lambda_1|A = (\lambda_2|A)^R$ then

- λ_1 starts with an element of $in(T_1) - A$, has elements of A in the middle,and ends with an element of $out(T_1)$, and
- λ_2 starts with an element of $out(T_2) - A$ and ends with an element of $out(T_2)$.

Hence the join of the cyclic ordering represented by λ_1 with the cyclic ordering represented by λ_2 does not satisfy $in(T)$ and $out(T)$. We conclude that the set (8) is empty. This completes the proof of correctness of the join procedure; it remains to prove the claim and to prove Lemma 4.13.

proof of claim. Assume without loss of generality that $flipped_{r_1}(T_1, T_1')$ is false, so the order of children of r_1 in T_1' is $a_1b_1c_1$. Hence the cyclic ordering σ_1 represented by $fr(T_1')$ has the form $(\alpha_1\beta_1\gamma_1)$ where α_1 consists of elements of $leaves(a_1) = in(T_1) - A$, β_1 consists of elements of $leaves(b_1) = A$, and γ_1 consists of elements of $leaves(c_1) = out(T_1)$. Suppose for a contradiction that $flipped_{r_2}(T_2, T_2')$ is true, so the order of children of r_2 in T_2' is $c_2b_2a_2$. Then the cyclic ordering σ_2 represented by $fr(T_2')$ has the form $(\gamma_2\beta_2\alpha_2)$, where γ_2 consists of elements of $leaves(c_2) = A$, β_2 consists of elements of $leaves(b_2) = out(T_2) - A$, and γ_2 consists of elements of $leaves(a_2) = in(T_2)$. Hence the linear ordering $\sigma_2|\overline{A}$ has the form $\beta_2\alpha_2$. Substituting $\sigma_2|\overline{A}$ for $\sigma_1|A$ in σ_1 yields σ_1 *join* $\sigma_2 = (\alpha_1\beta_2\alpha_2\beta_1)$, which is not (in,out)-reduced. Thus we have a contradiction. □

Before beginning the proof of Lemma 4.13, we prove two auxiliary lemmas:

Lemma 4.14 *Let T be a PQ-tree reduced and segregated with respect to $in(T)$ and $out(T)$. Let T' be the PQ-tree obtained from T by exchanging the root's two children $lca(in(T))$ and $lca(out(T))$. Then $C(T') = C(T)$.*

proof Before we prove lemma 4.14, note that it is not trivial. In particular, $L(T')$ is not necessarily the same as $L(T)$. The reason is that $lca(in(T))$ or $lca(out(T))$ (or both) may be an R-node, in which case merely exchanging the two children of the root is not a valid equivalence transformation.

In exchanging the two children of the root, we map every linear ordering $\alpha\beta$ generated by T to $\beta\alpha$ (where α is an ordering of $in(T)$ and β is an ordering of $out(T)$, or vice versa). Since $\alpha\beta$ and $\beta\alpha$ are valid representations of the same cyclic ordering, we have $C(T') = C(T)$. □

Lemma 4.15 *Let T be a PQ-tree reduced and segregated with respect to $in(T)$ and $out(T)$. Suppose that $lca(in(T))$ is an R-node and $lca(out(T))$ is not. Let T' be the PQ-tree obtained from T by renaming $lca(in(T))$ to be a Q-node. Then $C(T') = C(T)$.*

proof Clearly $C(T') \subseteq C(T)$; it remains to prove the other containment. Let $\tau' \in L(T')$,

and let $\alpha = \tau'|in(T), \beta = \tau'|out(T)$. By Lemma 4.12, there is a $\tau \in L(T)$ such that $\tau|in(T) = \alpha$ and $\tau|out(T)$ is either β or β^R. If the former, then τ and τ' represent the same cyclic ordering, and we are done. Otherwise, let T_1 be the PQ-tree equivalent to T such that $frT_1 = \tau$. Flip every node in $T_1|out(T)$ without flipping any other nodes of T_1; we can do this because $lca_{T_1}(out(T))$ is not an R-node. Let T_2 be the result. Then $frT_2|in(T) = \alpha$ and $frT_2|out(T) = \beta$, so frT_2 represents the same cyclic ordering as τ' represents. This proves $C(T') \subseteq C(T)$. □

proof of Lemma 4.13. Segregate T with respect to $in(T)$ and $out(T)$. Let $a = lca(in(T))$ and $d = lca(out(T))$. The root now has two children, a and d, so we may assume it is Q-node. Next, segregate $T|out(T)$ with respect to B and C, and let $b = lca(B)$ and $c = lca(C)$. Now d has two children, b and c, so we may assume it is a Q-node or an R-node.

We want b to be the left child of d; if it is not, flip d, a, and the root (we must flip all three nodes because d and a may be R-nodes). Next, we want a to be the left node of the root. If this is not the case, we apply Lemma 4.14 to exchange a and d as childdren of the root.

If d is an R-node, we eliminate the R-node as on page 18, attaching its children b and c to the root in its place. Then the root has children a, b, c satisfying the lemma. Suppose, therefore, that d is not an R-node, but a Q-node. Rename it to be an R-node, and, if a *is* an R-node, rename it to be a Q-node. By two applications of Lemma 4.15, it can be shown that this modification does not change the set of represented cyclic orderings. Now that d is an R-node, we proceed as above. □

5 Planar Graph Preliminaries

Lemma 5.1 *If a simple graph G with n nodes and m edges is planar, then $m \leq 3n - 6$.*

proof follows from Euler's formula for the number of faces of a planar graph. See, e.g., [5]. □

By Lemma 5.1, a planarity-testing algorithm can immediately reject a graph G that has more than $3n$ edges.

In an embedding in the plane, one face, called the exterior face, is an infinite region. To avoid this, we may consider embeddings in the surface of a sphere, for such an embedding can be mapped directly into an embedding on a plane, and vice versa (see, e.g., [5])

In [6], Edmonds described a combinatorial representation of a planar embedding of a graph G, namely a collection of cyclic orderings $\{\sigma_v \mid v \text{ is a node of } G\}$ of edges of G, where σ_v cyclically orders the edges incident to v. Given an embedding of a graph on the surface of a sphere, such a combinatorial embedding is found by observing the clockwise

ordering of edges around each node when the graph is viewed from a point outside the sphere.

Lemma 5.2 *[30] A graph G is planar iff its biconnected components are planar. Moreover, the combinatorial representation of a planar embedding of G can be immediately obtained from the combinatorial representations of planar embeddings of its biconnected components.*

Lemma 5.3 *[29] Any algorithm running on a concurrent-read, concurrent-write model of parallel computation can be simulated on a weaker model with exclusive-read and exclusive-write at a cost of $O(T(n))$ time per simulated step, where $T(n)$ is the time for sorting n elements on n processors.*

Remark: It is sufficient to sort integers of polynomial magnitude, so we can use, e.g. the algorithm of Subsection 2.2 to get $T(n) = O(\log n)$ (with a small multiplicative constant).

Lemma 5.4 *[28] A graph G on n nodes and m edges can be (edge) partitioned into its biconnected components in $O(\log^2 n)$ time on $n + m$ processors.*

Remark: The algorithm of [28] works in $O(\log n)$ time on a concurrent-write model of parallel computation. Using Lemma 5.3, we can run the algorithm of [28] on our model in $O(\log^2 n)$ time.

By Lemmas 5.2 and 5.4, we can assume without loss of generality that the graph to be tested for planarity is biconnected.

A graph G_1 is a *minor* of another graph G_2 if G_1 can be obtained from G_2 by deletion of nodes and edges, and contraction of edges. Thus, for example, any minor of a planar graph is planar, because the deletions and contractions can be carried out while maintaining an embedding of the graph on the sphere.

A *Jordan curve* is a simple closed curve. For example, the boundary of a face is a Jordan curve. A Jordan curve on the surface of the sphere separates the surface into two regions, such that any continuous path starting in one region and ending in the other must intersect the Jordan curve.

Suppose that the graph G is planar, and fix an embedding of G in the surface of a sphere. A Jordan curve in the surface of the sphere will be called an H-curve (in G) if

- one region contains all the nodes and edges in the fragment H and the other region contains all the nodes and edges in $G - H$, and
- the curve intersects each linking edge of H at exactly one point.

Let G be a connected graph with a connected subgraph H. We say that H is *bound in G* if $G - H$ is connected. We call an edge e of G *linking edge* of H if exactly one

endpoint of e is in H. That endpoint is called the edge's *linking point*, and the endpoint not in H is called the edge's *outside point*. Let $le(H)$ be the set of linking edges of H.

Suppose H is bound in G. For each embedding of G, we define a cyclic ordering the linking edges $le(H)$ around H, analogous to the cyclic ordering in Edmonds' scheme for representing an embedding. Fix an embedding of G, and consider the induced embedding of $G - H$. Because H is connected, there is a single face F of the embedding of $G - H$ in which all the nodes and edges of H are embedded. Otherwise, some edge of H would have to cross the embedding of G_H. Because $G - H$ is connected, the face F has a connected boundary. Choose a Jordan curve within F but closer to the boundary of F than any node of H. Then the curve defines two regions, one containing the nodes of H and the other containing the nodes of $G - H$, . The curve must intersect each linking edge of H, because each such edge goes from a node of x to the boundary of F. It can be chosen so that it intersects every linking edge of H exactly once. So chosen, it is an H-curve.

Consider a point traced along the H-curve. For uniqueness, we will require that the H-curve is traced in a clockwise direction; i.e. so that when viewed from outside the sphere, the region to the right of the direction of motion is the region containing the nodes of H. Such a point intersects the linking edges of H in some cyclic order. The cyclic orderings of the linking edges $le(H)$ determined in this way by embeddings of G will be called *embedding rotations* of H in G. (If G has no embeddings, then H has no embedding rotations in G.)

Lemma 5.5 *Let G' be a connected minor of G in which H has been contracted to a connected subgraph H' and no linking edges of H have been deleted. An embedding rotation of H in G is an embedding rotation of H' in G'.*

proof Let σ be any embedding rotation of H in G. Choose an embedding of G and an H-curve in accordance with σ. The contractions and deletions used to obtain G' from G may be carried out on the sphere in such a way that the cyclic ordering corresponding to the H-curve remains σ. □

Clearly the set of embedding rotations of a bound subgraph H in G are determined by the possible embeddings of G, and hence by the structure of G. The following lemma is a first step towards the determination of the set of embedding rotations of H from the structure of G.

Lemma 5.6 *Let G' and H' be as in Lemma 5.5. Suppose $G' - H'$ consists of only two nodes, x and y. Then the linking edges of H whose outside point in G' is x are all consecutive in any embedding rotation of H in G.*

proof By Lemma 5.5, we need only show that the edges with outside point x in G' are consecutive in any embedding rotation σ of H' in G'. Choose an embedding of G' and an

H'-curve in accordance with σ. Consider σ as a liner ordering starting at some linking edge of H' whose outside point is y. Let (h_1, x) and (h_2, x) be, respectively, the first and last edge in σ whose outside point is x. Since H' is connected, there is a path consisting of nodes of H from h_1 to h_2. Combine this path with the edges (h_1, x) and (h_2, x) to form a Jordan curve. Of the two regions R_1 and R_2 formed by this curve, only one (say R_2) contains the node y. Then any linking edge of H' with outside point y must lie in R_2, else it would intersect the Jordan curve.

We see that the portion of the H-curve in R_1 only intersects those linking edges of H that go to x. But note that the H-curve only enters and leaves R_1 once, else it would intersect (h_1, x) or (h_2, x) more than once. It follows that between (h_1, x) and (h_2, x) in the linear ordering derived from σ, only edges from H to x appear. That is, the collection of edges from H to x are a consecutive sequence in any embedding rotation of H in G. □
Note: In the following section, we make use of the notation that identifies a set of nodes with the subgraph induced by that set of nodes.

6 Our Efficient Parallel Planarity Testing Algorithm

Our main theorem is

Theorem 6.1 *A graph with n nodes can be tested for planarity in $O(\log^2 n)$ time using n processors. If the graph is planar, a combinatorial representation of a planar embedding can be found within the same bounds.*

The basic strategy of our planarity-testing algorithm is to process the graph "from the bottom up," starting with embeddings of individual nodes and ending with embeddings of the whole graph. A basic step in the algorithm is combining embeddings of subgraphs to form an embedding of the larger subgraph. We cannot merely choose a single embedding for each subgraph, for the embeddings of two subgraphs might be chosen to be inconsistent, preventing the embeddings from being combined. Instead, we use PQ-trees to represent the set of all embeddings of each subgraph. Once the planarity-testing algorithm succeeds, a "top-down" process can obtain a combinatorial embedding of the graph.

The first step of the algorithm is to find the biconnected components of the input graph, using the algorithm of Lemma 5.4. By Lemma 5.2, all that remains is to show how to test a biconnected graph G for planarity, and how to find a combinatorial embedding for G, if G is planar.

The second step is to find an *st-numbering* for G. An assignment of distinct integers from 0 to $n-1$ to the nodes of G is called an *st*-numbering[2] if two adjacent nodes s and t are assigned 0 and $n-1$ respectively, and every other node is adjacent to both a lower-numbered and a higher-numbered node. Note that an *st*-numbering of G induces a direction on the edges of G—namely, an edge points toward its higher-numbered endpoint. Accordingly, we call an edge incident to v an *incoming* edge if its other endpoint is numbered lower than v, an *outgoing* edge if its other endpoint is numbered higher than v. Let $in(v)$ be the set of incoming edges of v, and let $out(v)$ be the set of outgoing edges. The important fact about an *st*-numbering is that in the resulting directed acyclic graph, for every node v, there is a directed path from s to t through v.

Lemma 6.1 *[17] If G is biconnected, then G has an st-numbering.*

In [7], Even and Tarjan give a linear-time sequential algorithm for finding an *st*-numbering. This algorithm does not seem parallelizable. Fortunately, Maon, Schieber, and Vishkin have an efficient way to find an *st*-numbering in parallel.

Theorem 6.2 *[22] Given a biconnected graph G on n nodes and $m > 1$ edges, and an edge $\{s,t\}$, an st-numbering can be found in $O(\log^2 n)$ time on m processors.*

Remark: The algorithm of [22] works in $O(\log n)$ time on a concurrent-write model of parallel computation, but we can simulate the algorithm in $O(\log^2 n)$ time on an exclusive-write model, as in Lemma 5.4.

The remainder of our planarity algorithm may be viewed as a contraction process on the *st*-numbered graph, taking place over a series of stages. We start with the original *st*-numbered graph $G^{(0)} = G$. In the i^{th} stage, we choose a collection of edges of the graph $G^{(i)}$ in accordance with the *st*-numbering. We contract these edges, identifying their endpoints, and we update the *st*-numbering, producing the graph $G^{(i+1)}$. We stop after stage I if every node in $G^{(I+1)}$ except s and t is adjacent only to s and t.

After the i^{th} stage of the above contraction process, a node v in $G^{(i+1)}$ corresponds to a subgraph $H(v)$ of the original graph G; namely, $H(v)$ is the subgraph of G induced by those nodes identified to form v. Thus, if in stage i the node $v \in G^{(i+1)}$ was formed by identifying the nodes $v_1, \ldots, v_p \in G^{(i)}$, then $H(v) = H(v_1) \cup \cdots \cup H(v_p)$ (using our notation equating a set of nodes with the subgraph it induces). Generalizing slightly, if $v \in G^{(i)}$ and $j \leq i$, let $H^{(j)}(v)$ be the subgraph of $G^{(j)}$ whose nodes were identified to form v. For a node $v \in G^{(j)}$, let $v^{(i)}$ be the (unique) node of $G^{(i)}$ such that $v \in H^{(j)}$.

Note that $G^{(i)}$ is actually a multi-graph, not a graph. That is, $G^{(i)}$ may have multiple edges with the same endpoints. The reason is that two nodes adjacent to a common node

[2] As originally defined, an *st*-numbering ran from 1 to n, but we find it convenient to make this minor change.

u may have been identified to form a node v, in which case the node v will have two edges to u.

We choose our edges at each stage i so that

- Neither s nor t is ever identified with any other node; i.e. $s^{(i)} = s$ and $t^{(i)} = t$ for all i.
- For each node $v \neq s,t$ in G_i, the subgraph $H(v)$ permits a PQ-tree representation of the set of its embeddings.
- The st-numbering is easy to update, following contraction of edges.
- Only $O(\log n)$ stages are needed.

We first show how the edges are selected, then prove that our method of choosing edges has the above properties. Then we describe the method for representing the set of embeddings of a subgraph with a PQ-tree, and show how this representation is updated when edges are contracted. Finally, we show how to obtain an embedding of the original graph G.

We say a node $v \neq s,t$ of $G^{(i)}$ is *joinable* if v is adjacent to some node $u \neq s,t$ in $G^{(i)}$. We use a sequence of twelve stages, called a *phase*, to reduce the number of joinable nodes by a factor of two. In particular, we show that during a phase every joinable node is identified with some other joionable node. Thus, if $G^{(i)}$ is the graph immediately following the completion of a phase, then for any joinable node v of $G^{(i)}$, $|H^{(i-8)}(v)| \geq 2$. This shows that each phase reduces the number of joinable nodes by a factor of two, so only $\lceil \log n \rceil$ phases $= 12\lceil \log n \rceil$ stages are needed.

A phase has two parts, an *s-rooted* part and a *t-rooted* part, and each part consists of two subparts, a *main* subpart and a *clean-up* subpart. Each subpart consists of three stages.

For the s-rooted part of a phase, we construct a spanning tree of $G^{(i)} - \{t\}$ rooted at s. We show below that every node $v \in G^{(i)}$ other than s and t is adjacent to some lower-numbered node and to some higher-numbered node. For each such v, let its parent $p(v)$ be the highest-numbered neighbor of v whose number is less than that of v. We thereby define a "multi-tree," a graph that would be a tree if multiple edges were identified. The root of the multi-tree is s. Using parallel pointer-hopping, compute for each node v the distance distance from s to v in the multi-tree. Call a node "even" or 'odd," according to whether this distance is even or odd. In the main subpart, we contract all edges connecting even nodes to their (necessarily odd) parents. In the clean-up subpart, we contract all edges connecting odd leaves to their (necessarily non-leaf) parents, except for those leaves whose parent is s. In each case, we identify children with parent and

assign to the resulting node. Say a child is type 1 if its only lower-numbered neighbor is its parent, and type 2 otherwise. Below, we prove that if $G^{(i)}$ is planar, each parent has at most two type 2 children. In the first stage comprising a subpart, all type 1 children are simultaneously identified with their parents, contracting all the edges between them. In each of the second and third stages, a single type 2 child is identified with its parent, contracting the edges between them.

The t-rooted part of a phase is similar; the parent of v is chosen to be the lowest-numbered neighbor of v with a higher number than v.

Let f be the function assigning to each node its st-number. Note that if the i^{th} stage belongs to the s-rooted part of a phase, then for any $v \in G^{(i)}$, $f(v^{(i+1)}) \leq f(v)$. (If the i^{th} stage is part of the t-rooted part of a phase, then $f(v^{(i+1)}) \geq f(v)$.) Note also that the numbering of the nodes of $G^{(i)}$ is not, strictly speaking, an st-numbering, because there are numbers not assigned to any node, and the largest assigned number $(n-1)$ may in fact exceed the number of nodes in $G^{(i)}$. However, we will continue to refer to this numbering as an st-numbering, because it has the property of an st-numbering needed for our purposes: that every node other than s and t is adjacent to some lower-numbered node and to some higher-numbered node.

Lemma 6.2 *For any i, suppose that the numbering f of $G^{(i)}$ has the property that every node other than s and T has both an incoming and an outgoing edge. Then, the numbering f of $G^{(i+1)}$ has the same property.*

proof Let v be any node of $G^{(i+1)}$. Assume without loss of generality that the i^{th} stage belongs to the s-rooted part of a phase. Suppose $H^{(i)}(v) = \{v_1, \ldots, v_p\}$, where we asssume without loss of generality that v_1 was a parent and the other nodes (if any) were children. By the inductive hypothesis, v_1 has an incoming edge (u, v_1) in $G^{(i)}$, so there is an edge $(u^{(i+1)}, v)$ in $G^{(i+1)}$. We have $f(u^{(i+1)}) \leq f(u)$, $f(u) < f(v_1)$ by definition of an incoming edge, so $f(u^{(i+1)}) < f(v_1)$. Since $f(v_1) = f(v)$ by choice of v_1, we conclude that $(u^{(i+1)}, v)$ is an incoming edge of v. It remains to show that v has an outgoing edge. But v_p has an incoming edge (v_p, w) in $G^{(i)}$. There are two cases. If w was identified with its parent $p(w)$ in stage i, $f(w^{(i+1)}) = f(p(w)) > f(v_p)$ by choice of $p(w)$. If w was not identified with its parent in stage i, then $f(w^{(i+1)}) = f(w) > f(v_p) \geq f(v_p^{(i)})$. In either case, $(v_p^{(i+1)}, w^{(i+1)})$ is an outgoing edge of $v_p^{(i)} = v$. □

Recall that a child u of v in the s-rooted multi-tree is said to be of type 1 if its only lower-numbered neighbor is v. Similarly, a child u of v in the t-rooted multi-tree is said to be of type 1 if its only higher-numbered neighbor is v. If u is not of type 1, it is of type 2.

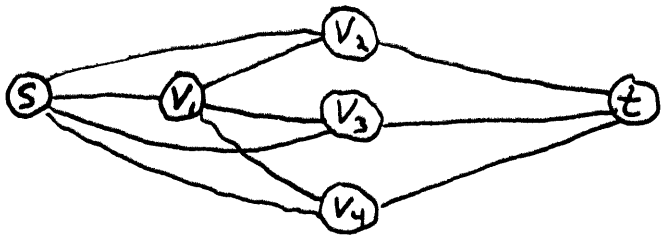

Figure 19: A non-planar graph, used in the proof of Lemma 6.3

Lemma 6.3 *Let $v_1 \neq s,t$ be a node of $G^{(i)}$ with three type 2 children v_2, v_3, v_4 in an s-rooted (or t-rooted) multi-tree. Then $G^{(i)}$ is not planar.*

proof By a series of contractions of edges and deletions of nodes and edges, we obtain from $G^{(i)}$ the graph depicted in Figure 19. The reader can check that by eliminating the edges $\{s,t\}$ and $\{s,v_1\}$, we obtain $K_{3,3}$, which is known not to be planar. Thus $G^{(i)}$ is not planar. □

We now consider the representation of the set of embeddings of a subgraph. Let $v \neq s,t$ be a node of $G^{(i)}$. It follows from Lemma 6.2 that by contraction of edges other than those incident to v and those between s and t, one can obtain from $G^{(i)}$ a graph $\Gamma(v)$ containing only the nodes s,t, and v in which all the incoming edges of v are incident to s and all the outgoing edges are incident to t. In $\Gamma(v)$, uncontract all those edges contracted to form v from $H(v)$. Let $\widehat{H}(v)$ denote the resulting graph.

Because $H(v)$ was formed by contraction of edges, $H(v)$ is connected. It follows that $H(v)$ is bound in $\widehat{H}(v)$. We call the embedding rotations of $H(v)$ in $\widehat{H}(v)$ the *arrangements* of v. Note that the arrangements of v are cyclic orderings of the edges incident to v. We let $\sigma(v)$ be the set of arrangments of v.

Note that by Lemma 5.5, we have

Lemma 6.4 *Any arrangement of v is an embedding rotation of v in $\Gamma(v)$.*

From Lemma 5.6, we may obtain

Corollary 6.1 *In any embedding rotation of v in $\Gamma(v)$ (and hence in any arrangement of v), the incoming edges are consecutive (and hence the outgoing edges are consecutive).*

proof Apply Lemma 5.6 with $G' = v$, $x = t$, and $y = s$. □

We represent the set $\sigma(v)$ of arrangements of v by an (in,out)-reduced PQ-tree $T(v)$ whose ground elements are the edges incident to v, and where $in(T(v))$ =the set of incoming edges of v and $out(T(v))$ =the set of outgoing edges of v.

If $v \in G^{(0)}$, then $H(v) = \{v\}$, so any cyclic ordering of the edges incident to v is an arrangement of v, provided that the incoming edges are consecutive and the outgoing edges are consecutive. In this case, therefore, we let $T(v)$ be the tree illustrated in Figure 20.

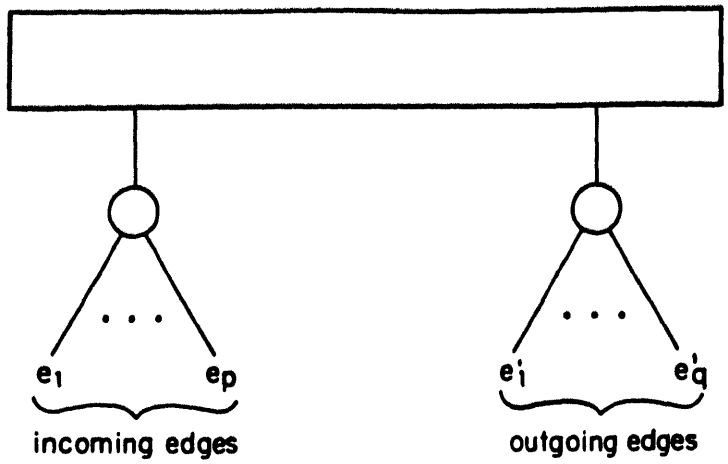

Figure 20: The PQ-tree $T(v)$ representing the set of arrangements of a node $v \in G^{(0)}$.

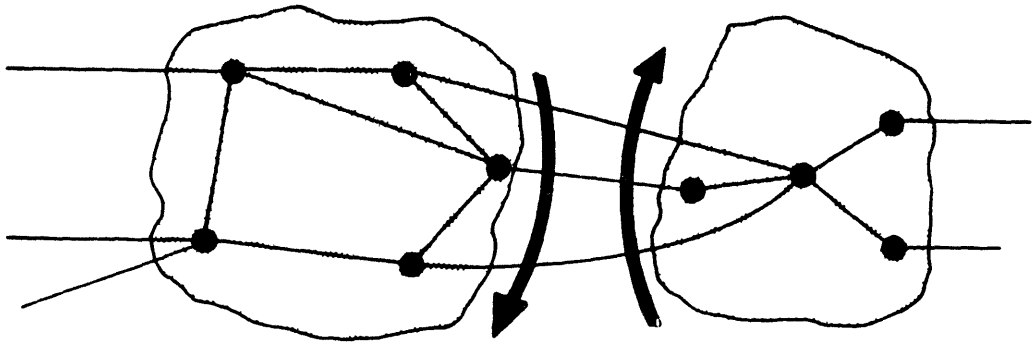

Figure 21: The order in which the edges between $H(u)$ and $H(v)$ are encountered in a clockwise contour around $H(u)$ is the reverse of the order in which they are encountered in a clockwise contour around $DH(v)$.

The invariant of the planarity algorithm is:

$$\text{For each node } v \neq s,t \text{ of } G^{(i)}, C(T(v)) = \sigma(v) \tag{10}$$

The base case $i = 0$ is satisfied if we use the construction of Figure 20. For each node $v \neq s,t$ in $G^{(i+1)}$, our algorithm computes the PQ-tree $T(v)$ from the PQ-trees $T(v_1)\ldots T(v_p)$, where $H^{(i+1)} = \{v_1,\ldots,v_p\}$. Below we describe this computation and show that, assuming (10) holds for i, it holds for $i+1$.

Consider contracting of the edges between just two nodes of $G^{(i)}$, say u and v, forming a node w of $G^{(i+1)}$. Let *shared*(u,v) be the set of edges between u and v. We need to compute the tree $T(w)$ from the trees $T(u)$ and $T(v)$.

Lemma 6.5 *Suppose the node $w \in G^{(i+1)}$ was formed by contracting all the edges shared(u,v) between the nodes $u,v \in G^{(i)}$. Suppose $C(T'(u))$ is contained in $\sigma(u)$ and contains all the embedding rotations of $H(u)$ in $\widehat{H}(w)$. Suppose $C(T'(v))$ is contained in $\sigma(v)$ and contains all the embedding rotations of $H(v)$ in $\widehat{H}(w)$. Assume that $T'(u)$ and $T'(v)$*

are reduced with respect to shared(u,v). *Let* T *be the (in,out)-reduced join of* $T'(u)$ *with* $T'(v)$. *Then* $\sigma(w) = C(T)$

Before giving the proof, we recall the definition of *join* from Subsection 4.5. For cyclic orderings $\sigma_1 \in C(T'(u))$ and $\sigma_2 \in C(T'(v))$, we let σ_1 *join* σ_2 denote the cyclic ordering obtained by substituting $\sigma_2|\overline{shared(u,v)}$ for $\sigma_1|\overline{shared(u,v)}$ in σ_2. The *(in,out)-reduced join* of $T'(u)$ with $T'(v)$ is a PQ-tree T such that

$$C(T) = \{\sigma_1 \textit{ join } \sigma_2 : \sigma_1 \in L(T'(u)), \sigma_2 \in L(T'(v)), \sigma_1|shared(u,v) = (\sigma_2|shared(u,v))^R \text{ and } \sigma_1 \textit{ join } \sigma_2 \text{ satisfies } in(w) \text{ and } out(w)\} \quad (11)$$

where $in(w) = in(u) \cup in(v) - shared(u,v)$ and $out(w) = out(u) \cup out(v) - shared(u,v)$.

proof of Lemma 6.5 Fix an element $\sigma \in \sigma(w)$. We show it belongs to the right-hand side of (11). Choose an embedding of $\widehat{H}(w)$ giving rise to the arrangement σ. The embedding defines an embedding rotation σ_1 of $H(u)$ in $\widehat{H}(w)$, and an embedding rotation σ_2 of $H(v)$ in $\widehat{H}$(w). By Lemma 6.4, $\sigma_1 \in \sigma(u)$ and $\sigma_2 \in \sigma(v)$. By an application of Lemma 5.6, the elements of $shared(u,v)$ are consecutive in σ_1 and σ_2. To show that $\sigma_1|shared(u,v) = (\sigma_2|shared(u,v))^R$ can be seen by inspecting Figure 21. By applying the definition of *join* , we see that $\sigma = \sigma_1$ *join* σ_2.

It is easy to see that the right-hand side of (11) is contained in $\sigma(w)$. The idea is that given σ_1 and σ_2 satisfying the conditions of the right-hand side of (11), embeddings for $\widehat{H}(u)$ and $\widehat{H}(v)$ are chosen and combined to form an embedding of $\widehat{H}(w)$. Then σ_1 *join* σ_2 is the arrangement corresponding to this embedding. Further details are omitted. □

We finally consider the implementation of the contractions occuring during a stage of the algorithm. Assume without loss of generality that stage i belongs to an t-rooted part, so edges between parent and child are incoming to parent and outgoing from child.

Suppose stage i is the first stage in its subpart. Then the contractions have the following form: given a node u with type 1 children $v_1, \ldots, v_p$, contract all the edges from these children to u, forming a node w of $G^{(i+1)}$. The first step is to reduce $T(u)$ with respect to the coloring that assigns the color c_k to the edges from v_k to u; let $T'(u)$ be the reduction. Because each v_k is of type 1, all the outgoing edges of v_k are incident to u. Hence we need not reduce the $T(v_k)$'s; we may let $T'(v_k) = T(v_k)$. The second step is to compute the (in,out)-reduced join of $T'(u)$ with the $T'(v_k)$'s, and let the join be $T(w)$. Because all the outgoing edges of each v_k are edges shared with u, we may use special case 1 of the (in,out)-reduced join described in Subsection 4.5. In that subsection, it was mentioned that in special case 1, a tree can be simultaneously joined with many. Hence we can join $T'(u)$ with all the $T'(v_k)$'s simultaneously. By a slight generalization of Lemma 6.5, the resulting PQ-tree $T(w)$ satisfies $C(T(w)) = \sigma(w)$. Thus in this case the invariant (10) is maintained.

Next, suppose stage i is either the second or third stage in its subpart. Then the contractions have the following form: given a node u with a type 2 child v, contract all the edges between u and v. There are two cases. If all the incoming edges of u are incident to v, we reduce $T(v)$ with respect to $\mathit{shared}(u,v)$, obtaining $T'(v)$, and then use special case 1 to join $T'(v)$ with $T(u)$. If not all of the incoming edges of u are in $\mathit{shared}(u,v)$, we reduce $T(u)$ with respect to $\mathit{shared}(u,v)$ and $\mathit{in}(u) - \mathit{shared}(u,v)$, obtaining $T'(u)$. Similarly, we reduce $T(v)$ with respect to $\mathit{shared}(u,v)$ and $\mathit{out}(v) - \mathit{shared}(u,v)$, obtaining $T'(v)$. Finally, we join $T'(u)$ with $T'(v)$, using special case 1 of the (in,out)-reduced join described in Subsection 4.5.

Let the result of the join be $T(w)$. If we could apply Lemma 6.5, it would follow that $C(T(w)) = \sigma(w)$, satisfying the invariant (10). All we need to show is that $C(T'(u))$ contains all the embedding rotations of $H(u)$ in $\widehat{H}(w)$ (and similarly for $C(T'(v))$). Certainly $C(T(u))$ contains all these embedding rotations, so it remains to prove that reducing $T(u)$ with respect to $\mathit{shared}(u,v)$ and $\mathit{in}(u) - \mathit{shared}(u,v)$ did not force out any embedding rotations.

To show that in every embedding rotation of $H(u)$ in $\widehat{H}(w)$ the edges $\mathit{shared}(u,v)$ are consecutive, we apply Lemma 5.6 as before. To show that in every embedding rotation of $H(u)$ in $\widehat{H}(w)$ the edges $\mathit{in}(u) - \mathit{shared}(u,v)$ are consecutive, we start from $\widehat{H}(w)$, contract $H(v) \cup \{s\}$ to a single node y, and apply Lemma 5.6.

We have shown how a stage may be implemented. To compute the joins completely takes $O(\log^2 n)$ time. However, as discussed in Subsections 4.3 and 4.5, we may compute a "provisional" join that is correct unless the correct result is the null PQ-tree in only $O(\log n)$ time. Hence to implement a stage quickly, we only compute provisional joins, This allows us to execute each phase in $O(\log n)$ time, for a total of $O(\log^2 n)$ time for all stages. We need not wait for the joins to be verified, because if a join fails to be verified, the graph is non-planar. We delay the verification of the joins until after the last stage is completed. Thus the time for all verifications merely adds $O(\log^2 n)$ time to the total time for the planarity-testing algorithm.

At each stage i, we assign one processor to each node of the PQ-tree $T(v)$, for every $v \in G^{(i)}$. The number of leaves of a PQ-tree $T(v)$ is just the number of edges incident to v in $G^{(i)}$, i.e. the degree of v. The sum of the degrees of all nodes in $G^{(i)}$ is at most the total number of edges in the original graph $G^{(0)}$. The total number of nodes in a PQ-tree is no more than twice the number of leaves, hence the number of processors needed is at most four times the number of original edges, or twelve times the number n of original nodes, by the remarks following Lemma 5.1. To reduce the number of processors to n, we simulate twelve processors by one, increasing the running time by a constant factor.

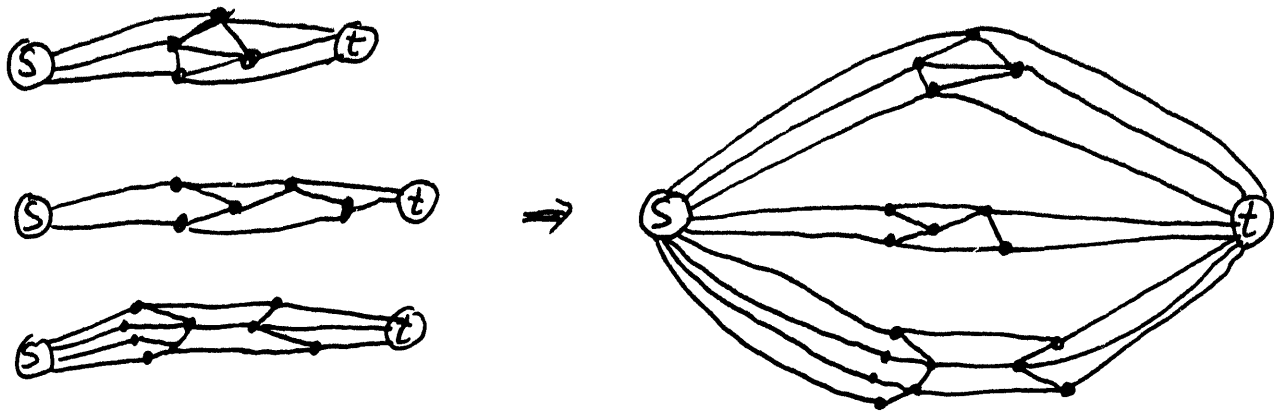

Figure 22: Combining embeddings of the $\widehat{H}(v)$'s (for $v \in G^{(I)}$) to obtain an embedding of $G^{(0)}$.

After the last stage, stage I, if for every node $v \in G^{(I)}$ the PQ-tree $T(v)$ is non-null, the graph is planar. For given an embedding of each $\widehat{H}(v)$, an embedding of $G^{(0)} = G$ can be obtained by identifying the nodes s and t appearing in each of the embeddings. The reason is that each v has no neighbors in $G^{(I)}$ but s and t. See Figure 22.

Finally, we sketch the method of obtaining a combinatorial embedding of each $\widehat{H}(v)$, assuming the planarity-testing algorithm successfully terminated. In the course of carrying out that algorithm, we defined a forest with trees rooted at the nodes $v \in G^{(I)}$, where the children of a node $u \in G^{(i+1)}$ are the nodes $H^{(i)}(v)$. The trees are all of height $I = O(\log n)$. Processing each tree from the top down, we choose orderings of the edges incident to each node $v \in G^{(i)}$, for $i = I, I-1, \ldots, 0$, that define a combinatorial embedding of $G^{(i)}$ of the sort considered by Edmonds. To choose an ordering of the edges incident to $v \in G^{(I)}$, read off the frontier of the tree $T(v)$. This can be done in $O(\log n)$ time using techniques of Subsection 2.1. We can then use small integer sorting to find the induced ordering on the edges incident to the nodes in $H^{(I-1)}$, inserting the orderings of the contracted edges (we can get these by reading the frontiers of the results of the PQ-tree intersections). We continue in this way, descending one level in $O(\log n)$ time, for a total of $O(\log^2 n)$ time. We end up with orderings on the edges incident to each node. The faces of the embedding are easily obtained from this representation using techniques of Subsection 2.1.

7 Conclusion

We have presented an efficient parallel algorithm for planarity. Natural questions arise: is there a more efficient parallel algorithm? Is there a faster parallel algorithm? To answer the first question, we observe that most, if not all, of the techniques used for the present algorithm can be made more efficient, at least asymptotically. In particular, all the

problems we discussed in Section 2 can be solved using randomized parallel algorithms in expected time $O(\log n)$ time using $n/\log n$ processors. Thus these problems may be solved very quickly in parallel using a number of processors that is within a constant factor of optimal. A careful study of our new planarity algorithm would probably yield the conclusion that, using these randomized techniques, planarity could be tested and an embedding found in $O(\log^2 n)$ time using $n/\log n$ processors.

Regarding the second question: We have made a major effort to speed up the planarity algorithm to run in $O(\log n)$ time on a concurrent-write model, but without success. It seems likely that, at least using $n \log n$ processors, this time bound could be achieved.

Our new result gives us hope that an $O(\sqrt{n})$ separator could be found for an n-node planar graph in polylogarithmic time using only n processors. Were this hope fulfilled, many other problems concerning planar graphs could be solved within these bounds, using divide-and-conquer techniques. Examples of such problems may be found in [19].

8 Acknowledgements

I would like to thank my colleagues at MIT's Theory of Computation Group, for their generous assistance. I would also like to thank John Reif and David Shmoys for their guidance.

This thesis is dedicated with love to Marcy Berger, who endured, encouraged, and enlivened the writing of it.

References

[1] Aho, A., J. Hopcroft, and J. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley (1974)

[2] Ajtai, M., J. Komlós, and E. Szemerédi, "An $O(n \log n)$ Sorting Network," *Proc. 15th Annual Symposium on the Theory of Computing* (1983), pp. 1-9

[3] Batcher, K. "Sorting Networks and Their Applications," *Proc. AFIPS Spring Joint Computer Conf.*, Vol. 32 (1968), pp. 307-314

[4] K. Booth and G. Lueker, "Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms," *Journal of Computer and System Sciences*, vol. 13, No. 3 (1976), pp. 335-379

[5] J. Bondy and U. Murty, *Graph Theory with Applications*, North-Holland:New York (1976)

[6] R. Edmonds, "A combinatorial representation for polyhedral surfaces," *American Mathematical Society Notices* 7 (1960), pp. 646

[7] S. Even and R. Tarjan, "Computing an *st*-numbering," *Theoretical Computer Science* 2 (1976), pp. 339-344

[8] S. Fortune and J. Wyllie, "Paralllelism in random access machines," *Proc. 10th Annual Symposium on the Foundations of Computer Science* (1978), pp. 98-108

[9] F. Harary and G. Prins, "The block-cutpoint-tree of a graph," *Publicationes Mathematica Debrecen* 13 (1966), pp. 103-107

[10] J. Hopcroft and R. Tarjan, "An efficient algorithm for graph planarity," *Journal of the ACM* 21 (1974), pp. 549-568

[11] J. Hopcroft and J. Wong, "Linear time algorithm for isomorphism of planar graphs," *Sixth Annual ACM Symposium on the Theory of Computing* (1974), pp.172-184

[12] J. Ja' Ja' and J. Simon, "Parallel algorithms in graph theory: planarity testing," *SIAM Journal of Computing* 11:2 (1982), pp. 313-328

[13] J. Ja'Ja' and S. R. Kosaraju, to appear

[14] Knuth, Donald G., *The Art of Computer Programming: Volume 3: Sorting and Searching*, Addison-Wesley, 1973

[15] Ladner, R. E. and M. J. Fischer, "Parallel Prefix Computation," *Journal of the ACM* Vol 27, No. 4 (1980), pp. 831-838

[16] Leighton, T., "Tight Bounds on the Complexity of Parallel Sorting," *Proc. 16th Annual Symposium on Theory of Computing* (1984), pp. 71-80

[17] A. Lempel, S. Even, and I. Cederbaum, "An algorithm for planarity testing of graphs," in *Theory of Graphs: International Symposium: Rome, July, 1966*, (P. Rosenstiehl, Ed.), Gordon and Breach, New York (1967), pp. 215-232

[18] R. Lipton and R. Tarjan, "A separator theorem for planar graphs," *SIAM Journal of Applied Math*, vol. 36, no. 2 (1979) pp. 177-189

[19] R. Lipton and R. Tarjan, "Applications of a planar separator theorem," *SIAM Journal of Computing*, vol. 9, no. 3 (1980) pp. 615-627

[20] L. Lovasz, "Computing ears and branchings in parallel," *26th Annual IEEE Symposium on Foundations of Computer Science* (1985) pp. 464-467

[21] B. Maggs, *Communication-Efficient Parallel Algorithms*, Masters' Thesis, MIT (1986)

[22] Y. Maon, Baruch Schieber, and Uzi Vishkin, "Parallel ear decomposition search (EDS) and *st*-numbering in graphs," Tel Aviv University Technical Report 46/86

[23] G. Miller and J. Reif, "Parallel tree contraction and its application," *26th Annual Symposium on Foundations of Computer Science* IEEE (1985) pp. 478-489

[24] Reif, John H. "An Optimal Parallel Algorithm for Integer Sorting," Proc. 26th IEEE Symposium on the Foundations of Computer Science, pp. 496-503

[25] Reif, J. H. and L. G. Valiant, "A Logarithmic Time Sort for Linear Size Networks," Proc. 15th Annual Symposium on the Theory of Computation, 1983, pp. 10-16

[26] Y. Shiloach and U. Vishkin, "An $O(\log n)$ parallel connectivity algorithm," *Journal of Algorithms* 3 (1982), pp. 57-67

[27] J. Smith, "Depth-first search for planar graphs," unpublished manuscript

[28] R. Tarjan and V. Vishkin, "Finding biconnected components and computing tree functions in logarithmic parallel time," 25th Annual Symposium on Foundations of Computer Science, IEEE (1984), pp. 12-22

[29] U. Vishkin, "Implementation of simultaneous memory access in models that forbid it," *Journal of Algorithms* 4, (1983) pp. 45-50

[30] Whitney, "Non-separable and planar graphs," *Trans. Amer. Math. Soc.* 34 (1930), pp. 339-362