Analysis and Design of Algorithms:
Double Hashing and Parallel Graph Searching

Mariko Molodowitch

Department of Information and Computer Science

University of California, Irvine

Technical Report 90-43

Dissertation

submitted in partial satisfaction of the requirements for the degree of

Doctor of Philosophy in Information and Computer Science

Dissertation Commmittee

Professor George S. Lueker, Chair

Professor Daniel S. Hirschberg

Professor Dennis J. Volper

1990

Dedication

This thesis is dedicated to my husband Dennis, without whose support it would never have been written.

Contents

List of Figures

Acknowledgements

I would like to thank the members of my committee, George Lueker, Daniel Hirschberg, and Dennis Volper, for all their help and guidance throughout my graduate career. Without their encouragement and support, this thesis would never have come to fruition. To my advisor, George Lueker, in particular, I am grateful for all his patience and wise counsel. I feel privileged to have worked with him.

The work on double hashing was joint work with George Lueker. It was presented at the *20th Annual ACM Symposium on Theory of Computing*, Chicago, IL, May, 1988. It will appear in substantially the same form in a future issue of *Combinatorica*. I am grateful to Nicholas Pippenger for a number of very helpful suggestions on the double hashing work, especially for suggesting the proof approach used in Chapter 2.

I thank the University of California for their fellowship support through the Faculty Mentor Program and the Graduate Opportunity Dissertation Fellowship Program, and to the National Science Foundation for support through Grant CCR 8912063.

Curriculum Vitae

Mariko Molodowitch

1969	B.A. in Physics, Radcliffe College, Harvard University
1988	M.S. in Information and Computer Science, University of California, Irvine
1990	Ph.D. in Information and Computer Science, University of California, Irvine
	Dissertation: *Analysis and Design of Algorithms Double Hashing and Parallel Graph Searching*
	Committee Chair: Professor George S. Lueker

Publications

D.S. Hirschberg, L.L. Larmore, M. Molodowitch, "Subtree Weight Ratios for Optimal Binary Search Trees," University of California, Irvine Technical Report No. 86–02, 1986.

G. Lueker and M. Molodowitch, "More Analysis of Double Hashing," *Proceedings of the Twentieth Annual ACM Symposium on Theory Of Computing.* pp. 354–359. May, 1988. To appear in *Combinatorica.*

Fields of Study

Major Field: Information and Computer Science

Studies in Algorithms and Parallel Computation

Abstract of the Dissertation

Analysis and Design of Algorithms:

Double Hashing and Parallel Graph Searching

by

Mariko Molodowitch

Doctor of Philosophy in Information and Computer Science

University of California, Irvine, 1990

George S. Lueker, Chair

The following is in two parts, corresponding to the two separate topics in the dissertation.

Probabilistic Analysis of Double Hashing

In [GS78], a deep and elegant analysis shows that double hashing is asymptotically equivalent to the ideal uniform hashing up to a load factor of about 0.319. In this paper we show how a resampling technique can be used to develop a surprisingly simple proof of the result that this equivalence holds for load factors arbitrarily close to 1.

Parallel Depth First Search of Planar Directed Acyclic Graphs

In 1988, Kao [Kao88] presented the first NC algorithm for the depth first search of a directed planar graph. Recently, Kao and Klein [KK90] reduced the number of processors required from $O(n^4)$ to linear, but the time bound is $O(\log^8 n)$.

We present an algorithm for the depth first search of a planar directed acyclic graph with k sources using $O(n)$ processors and $O(\log k \log n)$ time on a CRCW PRAM model. For planar dags with a single source and a single sink, we present a simple optimal algorithm which gives the depth first search in $O(\log n)$ time with $O(n/\log n)$ processors on an EREW PRAM. For a single-source multiple-sink planar dag, we have an $O(\log n)$ time $O(n)$ processor EREW algorithm. The EREW algorithms assume that the embedding is given. A simplified variant of the depth first search of a multisource planar dag can be used to solve the single source reachability problem for a planar directed acyclic graph in $O(\log^2 n)$ time

and $O(n)$ processors on an CRCW PRAM. Since an $O(\log^4 n)$ algorithm for this problem is used as a subroutine by Kao and Klein in their depth first search for the general planar directed graph, this will lower their time bound by a factor of $\log^2 n$. Our work uses the concept of a planar Euler tour depth first search, a depth first search in which the Euler tour around the tree is planar and crosses no tree edge. This concept may prove to be of use in other parallel algorithms for planar graphs.

Introduction

Since the introduction of computers, the quest for fast solutions to problems using this tool has been pursued in the two complementary areas of hardware and software. In software, a major thrust has been the search for algorithms guaranteed to produce solutions quickly to a specific problem for a wide range of instances. Sometimes, an algorithm which takes a long time on certain instances of input can be guaranteed to run faster on average, assuming a reasonable distribution of inputs. Another way to get a fast algorithm is to take advantage of new techniques made possible by advances in hardware. The use of parallel processors sometimes makes it possible to solve a problem more quickly than with a single processor.

This dissertation is in the two areas of distributional analysis of algorithms and design of parallel algorithms. In the area of analysis, we focus on the well-known technique of double hashing for record management. In the area of parallel algorithm design, we look at the problem of depth first search of directed planar graphs with no cycles. The following is a description of the two areas and the specific problems addressed in this work.

Distributional analysis of algorithms

Given an algorithm, one possible measure of its efficiency is the amount of time it takes to run the algorithm as a function of the size of the input. For example, suppose the input consisted of n items. If the algorithm spent a constant amount of time for each item, the functional dependence would be linear in n. During the early days of computer science, researchers found that there were many problems for which they could not find any algorithm guaranteed to run in time polynomial in the size of the input; in order to produce a correct answer in all cases, the algorithms would spend time exponential in the size of the input. In the early 1970's, the idea of NP-completeness was developed. The class of problems for which polynomial time algorithms exist became known as P. The class of problems for which, if the answer were somehow guessed or given, its correctness could be checked using a polynomial time algorithm, became known as NP. The work of Cook and Karp, followed by others, showed that there was another class of problems, named the NP-complete problems, such that the problems were all in NP and if any NP-complete problem could be shown to be in P, the class P and the class NP could be shown to be the same. [Coo71, Kar72].

Although finding a polynomial time solution to these problems has proven to be extremely hard, if not impossible, many NP-complete problems have algorithms which work quite well in practical cases. For such algorithms, we can use probabilistic distributional analysis: instead of assuming that the input is the worst case possible for the algorithm, we assume that the inputs are drawn at random from a specified probability distribution [Kar86]. If the specified probability distribution is close to that

found in real-life, or if the analysis is valid for a broad class of distributions, the results of the analysis can be useful.

This type of analysis can be used with problems other than NP-complete ones to find the behavior of a given algorithm under the assumption of certain input probability distributions. For example, suppose Algorithm A ran in n^2 time in the worst case, and Algorithm B ran in n^3 time in the worst case. However, suppose probabilistic analysis of the two algorithms showed that Algorithm A ran in n^2 time even in the average case, but that Algorithm B ran in linear time on the average. Then Algorithm B may be more useful than Algorithm A.

The algorithm we analyze in this way is double hashing, one of a class of techniques called hashing, used for managing a table of records. Given a key such as a name or ID number, a hash function gives the location of the record in a table. This can result in faster storage and retrieval times than for sequential tables where the records are stored in order of their keys. However, two keys can hash to the same location, resulting in a collision. Among the techniques used to solve this are linear probing, uniform hashing, and double hashing. Linear probing is simple to implement, but a probabilistic analysis shows that the extra time needed due to the collisions grows rapidly as the table becomes more and more full [Knu73]. Uniform hashing is a theoretical method, impractical for actual use, but analysis shows that the extra time needed does not grow as fast as for linear probing. Double hashing can be implemented, and has been shown empirically to behave like uniform hashing. A paper by Guibas and Szemeredi [GS78] gave a very difficult distributional analysis to explain this behavior of double hashing algorithms for tables up to about one-third full. The work presented in this thesis, done jointly with George Lueker, gives a simple proof for this behavior for tables arbitrarily close to being full. An earlier version of this work was presented at the *20th Annual ACM Symposium on Theory of Computing*, Chicago, IL, May 1988 [LM88]. It will appear in substantially the same form in a future issue of *Combinatorica*.

Parallel algorithms and depth first search

With the advent of new technology, computers with multiple processors have become more realistic, and interest has grown in algorithms which use processors operating in parallel. Parallel algorithms have been found which significantly reduce the time to solve a problem when compared to sequential algorithms using a single processor. With such parallel algorithms, not only the time to run the algorithm as a function of the size of the input, but also the number of processors used, becomes important. In particular, NC (Nick's Class, named after Nicholas Pippenger for an early significant paper in this area [Pip79]) is the class of problems which have parallel algorithms to solve them with specific bounds on both time and processor count. For an input of size n, the algorithm must have a polylog time bound: the time is bounded by $f(\log n)$ where f is a polynomial function. The algorithm also must use no more than $g(n)$ processors where g is a polynomial function. Much work in the area of parallel computation has been in showing that a problem is in NC, and in reducing the time and processor count once it is known to be in NC.

Depth first search of a directed graph G

(* This is a sequential algorithm for depth first search of a graph G with a set of vertices V and set of edges E. *)

```
Mark every vertex in the graph as unvisited;
While there exists a vertex which is marked unvisited do
   begin
      pick a vertex v which is marked unvisited;
      run procedure dfs(v);
   end


procedure dfs(v)
   Visit and mark v as visited;
   For all vertices w on v's adjacency list do
      if w is marked unvisited
         then run procedure dfs(w)
```

Figure 1
A sequential depth first algorithm for directed graphs

Sequential depth first search of directed and undirected graphs is a technique used as a basis for other sequential graph algorithms [AHU74, Eve79, BB88]. A sequential algorithm for depth first search of a directed graph is given in Figure 1. If parallel depth first search algorithms can be found with low time and processor count, they may be used as subroutines in other parallel graph algorithms just as in the sequential case. We review some of the work done in this area in Chapter 1 in Part II.

Kao and Klein [KK90] have recently presented a parallel algorithm for depth first search of planar directed graphs using a linear number of processors but with an $O(\log^8 n)$ time bound. In this dissertation we present new parallel algorithms with reduced time bounds for depth first search of planar directed graphs with no cycles. We also present a new parallel algorithm for the problem of finding all vertices reachable from a single vertex in a planar directed acyclic graph. Since Kao and Klein uses the solution to this problem as a subroutine in their depth first search, we cut their time bound by a factor of $\log^2 n$.

PART I
More Analysis of Double Hashing

CHAPTER 1
Scope of present work

In [GS78] a deep and elegant analysis showed that double hashing was equivalent to the ideal uniform hashing up to a load factor of about 0.319. In this dissertation, we give an analysis which extends this to load factors arbitrarily close to 1. We understand from [Kom86, Gui87] that Ajtai, Guibas, Komlós, and Szemerédi obtained this result in the first part of 1986; our analysis is of interest nonetheless because we demonstrate how a resampling technique can be used to obtain a remarkably simple proof.

A *hash table* will consist of an array of m *slots*, indexed from 0 to $m-1$, each of which can contain a key. The hash table *configuration* is the set of indices of the filled slots. The *load factor* of a table is the fraction of the slots which have been filled. Assuming we are using double hashing (see [Knu73] for definitions), let $C'_\alpha(m)$ be the average probe length during an unsuccessful search of a table of size m filled to load factor α. As in [GS78], this will include the probe that found an empty table slot.

It has long been known that for a theoretical model called *uniform hashing*, the average probe length is $(1-\alpha)^{-1} + O(m^{-1})$; see [Knu73] for definitions and a history of research on this problem. A number of papers [Ull72, AKS78, Yao85] have shown that uniform hashing is optimal among open addressing schemes in certain senses. Ullman [Ull72] showed that no hash function could consistently outperform uniform hashing for average insertion cost. In [AKS78], it was shown that for a class of functions called single-hashing functions, uniform hashing was optimal in retrieval costs. Yao [Yao85] generalized the proof to all open addressing hashing algorithms, but left open the question of lower bounds for insertion costs.

The term *hash pair* will refer to a pair $(h_1(K), h_2(K))$ where K is the key to be inserted, h_1 is the primary hash function, and h_2 is the secondary hash function which is used as a probe decrement. As in [GS78], we assume that

$$\Pr\{(h_1(K), h_2(K)) = (i,j)\} = \frac{1}{m(m-1)}$$

for all (i,j) with $0 \le i \le m-1$ and $1 \le j \le m-1$. Let $P^A_{\alpha,m}(k)$ be the probability that the probe length during an unsuccessful search of a table, which has been filled to a load factor α using algorithm A, is at least k. UH will stand for uniform hashing and DH for double hashing. So that $P^{\mathrm{A}}_{\alpha,m}(k)$ is defined when αm is not an integer, extend it by

$P^A_{\alpha,m}(k) = P^A_{\lfloor \alpha m \rfloor/m,m}(k)$. Note that

$$C'_\alpha(m) = \sum_{k=1}^{m} P^A_{\alpha,m}(k). \tag{1.1}$$

To minimize the amount of notation in the paper, we will adopt the following convention (though we will avoid the use of this terminology in the statement of theorems). The constant c will refer to a constant whose value will be left free. We will say that a function $f(m,c)$ is c-polysmall if for all positive p, for large enough c the function is $O(m^{-p})$. Note that

the sum of polynomially many functions which are c-polysmall is again c-polysmall,

if we can assert that $f(m,c)$ is c-polysmall except for small m, the phrase "except for small m" may be dropped, and

if $f(m,2c)$ is c-polysmall, then so is $f(m,c)$.

Also, throughout the paper we will let α be some arbitrary but fixed constant in the range $0 < \alpha < 1$, and let m range only over prime values.

Our goal is to prove that double hashing is asymptotically equivalent to uniform hashing for load factors arbitrarily close to 1. In fact, we can show that the distribution of the number of probes in an unsuccessful search is close to that obtained with uniform hashing.*

Theorem 1. *For each fixed $\alpha \in (0,1)$ and each $p \geq 1$, we can choose a constant c so that if $\delta = cm^{-1/2}\log^{5/2} m$, then*

$$P^{\mathrm{UH}}_{(1-\delta)\alpha,m}(k) + O(m^{-p}) \leq P^{\mathrm{DH}}_{\alpha,m}(k) \leq P^{\mathrm{UH}}_{(1+\delta)\alpha,m}(k) + O(m^{-p}),$$

where the hidden constants in the O-notation are independent of k and m, and m is restricted to assume only prime values.

By equation (1.1) this yields

Corollary 1. *For each fixed α with $0 < \alpha < 1$,*

$$C'_\alpha(m) = \frac{1}{1-\alpha} + O(m^{-1/2}\log^{5/2} m).$$

Proof Sketch. Let $p \geq 2$ in Theorem 1. Then by equation (1.1)

$$C'^{UH}_{(1-\delta)\alpha}(m) + O(m^{-1}) \leq C'^{DH}_{\alpha}(m) \leq C'^{UH}_{(1+\delta)\alpha}(m) + O(m^{-1}).$$

Since $C'^{UH}_\alpha = (1-\alpha)^{-1} + O(m^{-1})$, we can use the following two inequalities to complete the proof.

$$\frac{1}{1-(1-\delta)\alpha} \geq \frac{1}{1-\alpha} - \frac{\delta}{(1-\alpha)^2}$$

* We thank [SS90] for pointing out that this was implicit in an earlier version of this paper.

$$\frac{1}{1-(1+\delta)\alpha} \leq \frac{1}{1-\alpha} + \frac{\delta}{(1-\alpha)^2}$$

The second inequality will hold when m is sufficiently large so that $\delta \leq (1-\alpha)^2\alpha^{-1}$.

∎

The implied lower bound of this corollary is of course not very surprising, especially in view of [Ull72, AKS78, Yao85].

For any table configuration, for every empty slot x, let $\hat{\xi}(x)$ be the number of hash pairs which would cause x to be the next slot filled. As in [GS78], we can readily use this to express the probability $\xi(x)$ that x is the next position filled using double hashing:

$$\xi(x) = \frac{\hat{\xi}(x)}{m(m-1)}. \tag{1.2}$$

A key technique in our paper is the modification of the distribution of table configurations in a way which a) dominates the distribution that would be obtained by the original algorithm except with very small probability (in a sense made more precise later), b) forces the table to be equivalent to one obtained by uniform hashing, and c) causes only a very small change in table performance. The technique is similar in principle to a resampling technique used in [KK82]. There, a distribution which was nearly uniform was converted into a truly uniform distribution by a sampling procedure which rejected a few points. In our case, we will produce a table equivalent to uniform hashing by carefully adding a few extra items to the hash table. These extra points will be colored red, while the original items will be colored green. Our addition of red points is in some ways similar to the randomization proof strategy used in [AKS78, Yao85].

A useful fact is given by

Lemma 1 [Hoe63]. *Let X be the binomially distributed random variable giving the number of successes in n Bernoulli trials each having success probability p_0. Then for $\beta \geq 0$,*

$$\Pr\{X \geq (p_0+\beta)n\} \leq \exp(-2n\beta^2),$$

and

$$\Pr\{X \leq (p_0-\beta)n\} \leq \exp(-2n\beta^2).$$

In section 2 we observe that if a table is partially filled according to uniform hashing, but then we insert one more key according to double hashing, the distribution of the location into which the key will be inserted is, with high probability, nearly uniform. In section 3 we show how to use resampling to obtain a remarkably simple proof of Theorem 1.

CHAPTER 2
The distribution of $\hat{\xi}(x)$ in the uniform case

In this section we note that if we have a table which has been partially filled by uniform hashing, placing the next point by double hashing is very nearly the same as placing it by uniform hashing, except with small probability. This sort of result is implicit in [GS78], and we make considerable use in this section of the insights and machinery of that paper. Using an elegant approach suggested to us by [Pip88], we give a rather simple proof of the precise form (Lemma 3) we will want.

Lemma 2. *Let $0 \leq p \leq (2+\alpha)/3$. Suppose we construct a hash table configuration of size m by letting each slot, independently, be filled with probability p. Then there exists a constant c such that for every empty slot x,*

$$\hat{\xi}(x) \leq \frac{m}{1-p} + (c \log m)^{5/2} m^{1/2}$$

except with probability which is c-polysmall.

Proof. We will use the notion of progressions similar to those used in [GS78]. Call the set of slots $\{x+ld,\ x+(l-1)d,\ x+(l-2)d, \ldots,\ x+d\}$, where the numbers are mod m and indicate slot indices, the *potential progression of length l generated by stride d coming to x*. If all slots in this potential progression are filled, we can omit the word potential and say it is a *progression of length l generated by stride d coming to x*. Thus for fixed l and x, each stride generates just one potential progression, and actually generates a progression with probability p^l under the probability distribution of this lemma. (Note that a progression of length l coming to x contains progressions of length $l-1, l-2, \ldots, 3, 2$, and 1 coming to x.) If we adopt the convenient fiction that the $m-1$ progressions of length 0 coming to x generated by the strides $\{0, 1, \ldots, m-1\}$ are distinct, we can set up a correspondence between progressions of length $l \geq 0$ generated by strides d and hash pairs $(x+ld, d)$ which would cause x to be probed. Since we defined $\hat{\xi}(x)$ as the total number of hash pairs which would cause x to be probed, $\hat{\xi}(x)$ will be equal to the total number of progressions coming to x.

Now fix a table slot x, and let

$$k = c \log m. \tag{2.1}$$

For $0 \leq l \leq m-1$ let M_l be the number of progressions of length l coming to x. Note that by our above convention $M_0 = m-1$, and that M_l is a non-increasing function of l. Much as in [GS78, p. 254] we note that if the number of progressions coming to x of length k is at most M_k, then there are no progressions of length $(M_k+1)k$ coming to x; this is because even one progression of length $(M_k+1)k$ (with stride d) would contain M_k+1 progressions of length k coming to x (with strides d, $2d$, $\ldots$, $(M_k+1)d$). Because M_l is nonincreasing, M_l for $k < l \leq (M_k+1)k - 1$ is bounded by M_k. Hence, the total

number of progressions of length greater than k coming to x is bounded by $M_k^2 k$. From this it follows that we can bound the total number of progressions coming to x as follows:

$$\hat{\xi}(x) \leq kM_k^2 + \sum_{l=0}^{k} M_l. \tag{2.2}$$

We now fix x and l, where $l \leq k$, and seek to estimate M_l, which is the number of strides d which generate a progression of length l coming to x. For the present assume that

$$l \geq 2. \tag{2.3}$$

Note that there are $m-1$ possible strides, and that by the probability distribution in this lemma it is clear that each gives a progression of length l coming to x with probability p^l. Thus we might hope to use Lemma 1 to control the value of M_l, but there is of course a problem: while each slot is filled independently, the slots used by various strides are not disjoint. An elegant solution to this problem, from [Pip88], involves the use of a simple greedy method to partition the strides into subsets of which no two generate common slots.

First let us say that two strides d and d' *interact* if the two potential progressions of length l which they generate intersect, i.e., if

$$\begin{aligned} &\{x + ld, x + (l-1)d, x + (l-2)d, \ldots, x + d\} \\ &\cap \{x + ld', x + (l-1)d', x + (l-2)d', \ldots, x + d'\} \neq \emptyset. \end{aligned} \tag{2.4}$$

If they do not interact we will say they are *independent*. Note that the events specifying the presence of these two progressions are independent in the probabilistic sense if and only if the strides are independent in the sense just defined. One readily establishes that for any d,

$$|\{d' : d \text{ and } d' \text{ interact}\}| \leq l^2. \tag{2.5}$$

To see this, note that if one fixes d, chooses one element from each of the two sets in equation (2.4), and equates these elements, then there is a unique solution for d'—but there are exactly l^2 ways to choose one element from each set.

Let *GreedySelect*(S, r) be a procedure which extracts a set of r independent strides from a set S of strides according to the natural greedy method: It starts with an empty set and iteratively adds any stride which does not interact with those added so far. Since by equation (2.5) we know that no stride eliminates more than l^2 strides, we see that this process will succeed in constructing the desired set if $r \leq |S|/l^2$. We now decompose the initial set of strides according to the procedure *GreedyPartition* given in Figure 1.

A simple induction establishes that all of the *GreedySelect* calls succeed. Note that we are assured that the procedure will halt (for large m) since one establishes without much difficulty that

$$\sum_{i=0}^{\infty} \left\lfloor \frac{m-1}{l^2} \left(1 - \frac{1}{l^2}\right)^i \right\rfloor = m - O(l^2 \log m),$$

```
procedure GreedyPartition;
begin
  S := {1, 2, ..., m − 1}; i := 0;
  while |S| ≥ log^4 m do
    begin
      assert |S| ≥ (m − 1)(1 − 1/l^2)^i;
      D_i := GreedySelect(S, ⌊((m − 1)/l^2)(1 − 1/l^2)^i⌋);
      S := S − D_i;
      i := i + 1;
    end;
  N := i − 1;
  D' := S;
end;
```

Figure 1
Procedure *GreedyPartition*

and $l^2 \log m = o(\log^4 m)$. Also note that

$$\begin{aligned} &\{D', D_0, D_1, \ldots, D_N\} \text{ is a partition of } \{1, 2, \ldots, m-1\}, \\ &|D_i| = \left\lfloor \frac{m-1}{l^2}\left(1 - \frac{1}{l^2}\right)^i \right\rfloor, \\ &\text{All elements of } D_i \text{ are independent, and} \\ &|D'| \le \log^4 m \end{aligned} \tag{2.6}$$

We now use the independence of the D_i to produce the desired bounds. For convenience let

$$\gamma = \sqrt{1 - \frac{1}{l^2}}, \tag{2.7}$$

so

$$|D_i| = \lfloor \frac{m-1}{l^2} \gamma^{2i} \rfloor. \tag{2.8}$$

Let $P_0, P_1, P_2, \ldots, P_N, P'$ (respectively) be the number of strides in $D_0, D_1, D_2, \ldots, D_N, D'$ (respectively) which actually generate progressions, so that

$$M_l = P' + \sum_{i=0}^{N} P_i. \tag{2.9}$$

Now we know that each of the P_i is binomially distributed, and we can use Lemma 1 to assert that for each i, if we choose

$$\beta_i = \sqrt{c \log m} / \sqrt{|D_i|},$$

then

$$\Pr\{P_i \geq (p^l + \beta_i)|D_i|\}$$

is c-polysmall. (Note that this argument does not apply to D', because the necessary independence is lacking, but D' is small enough that this will not cause a problem.) Hence by equation (2.9) we can assert that, except with c-polysmall probability,

$$\begin{aligned}
M_l &= P' + \sum_{i=0}^{N} P_i \\
&\leq |D'| + \sum_{i=0}^{N} p^l |D_i| + \sum_{i=0}^{N} \beta_i |D_i| \\
&\leq |D'| + p^l \sum_{i=0}^{N} |D_i| + \sum_{i=0}^{N} \sqrt{(c \log m)|D_i|} \\
&\leq \log^4 m + p^l m + \sum_{i=0}^{N} \sqrt{cm \log m}\gamma^i / l \\
&\qquad \text{(by equation (2.6) and equation (2.8))} \\
&\leq \log^4 m + p^l m + \frac{\sqrt{cm \log m}}{l} \frac{1}{1-\gamma} \\
&\leq \log^4 m + p^l m + 2l\sqrt{cm \log m}, \qquad (2.10)
\end{aligned}$$

where the last step uses the fact that from equation (2.7) we have $1/\big(l(1-\gamma)\big) \leq 2l$. We can now drop the assumption equation (2.3) since if $l = 0$ then $M_l = m - 1$ so equation (2.10) is trivial, and if $l = 1$ then M_l is just the number of filled slots so equation (2.10) holds except with c-polysmall probability directly from Lemma 1.

Because M_k is squared in equation (2.2), we will need a stronger bound on it than is provided by equation (2.10), but that is readily obtained. For each stride d, the probability that it yields a progression of length k is p^k, which is c-polysmall since $p < (2+\alpha)/3 < 1$ and $k = c \log m$. Hence, the probability that M_k exceeds 0 is c-polysmall.

By equation (2.2) we now conclude that, except with c-polysmall probability, for all x

$$\begin{aligned}
\hat{\xi}(x) &\leq kM_k^2 + \sum_{l=0}^{k} M_l \\
&\leq 0 + \sum_{l=0}^{k} \big(\log^4 m + p^l m + 2l(cm \log m)^{1/2}\big) \\
&\leq \log^5 m + \frac{m}{1-p} + (c \log m)^{5/2} m^{1/2} \\
&\qquad \text{(where we have used equation (2.1))}
\end{aligned}$$

$$\leq \frac{m}{1-p} + (2c \log m)^{5/2} m^{1/2}$$

where the last step holds for large m. ∎

Lemma 3. *let $0 \leq \eta \leq (1+\alpha)/2$. Suppose we construct a hash table configuration of size m by adding ηm points according to uniform hashing. Then*

$$\hat{\xi}(x) \leq \frac{m}{1-\eta}\left(1 + \frac{(c \log m)^{5/2}}{m^{1/2}}\right)$$

except with probability which is c-polysmall.

Proof sketch. If we pick $p = \eta(1 + cm^{-1/2} \log m)$, then the distribution of Lemma 2 will include at least ηm points except with probability which is c-polysmall. ∎

CHAPTER 3
Proof of the Theorem

We now show how the Lemma 3 of the previous section combined with a sampling technique can be used to give a simple proof of Theorem 1.

The following definition and observation will be crucial. We say that configuration C_1 *dominates* configuration C_2 if $C_1 \supseteq C_2$. Then as observed in [GS78, p. 255], double hashing has the following *monotonicity* property. If C_1 dominates C_2, and we insert a key into C_1 (respectively C_2) to obtain C_1' (respectively C_2'), then C_1' dominates C_2'. In particular, this means that if we occasionally add a fictitious extra point to the table, this will never cause some slot to remain empty that would otherwise have been filled.

Proof of Theorem 1. Let

$$\delta = \frac{(c \log m)^{5/2}}{m^{1/2}}. \tag{3.1}$$

To obtain the right inequality in the theorem, we will show that

$$P^{\mathrm{DH}}_{\alpha,m}(k) \leq P^{\mathrm{UH}}_{(1+2\delta)\alpha,m}(k) + (\text{terms which are } c\text{-polysmall}) \tag{3.2}$$

(By the properties observed above of the c-polysmall notation, the factor of 2 on δ is unimportant.)

Consider the procedure *UsuallyDoubleHash* given in Figure 2, for inserting points into the table. Let $K_1, K_2, \ldots, K_n$, where $n = \lfloor \alpha m \rfloor$, be the keys to be inserted. (We will call these the *original keys* to avoid possible confusion with extra points to be added.) Usually an inserted point will be one of the original keys (these will be colored green), but with small probability it will be an extra point (colored red) which we add to facilitate the analysis. We henceforth let UDH stand for UsuallyDoubleHash.

In order to make the structure of the randomized resampling argument clear, we now describe the probability spaces which will be used. Fix n and m. The sequence of original keys defines a sequence of hash pairs $((i_1, j_1), (i_2, j_2), \ldots, (i_n, j_n))$ with $i_l = h_1(K_l)$ and $j_l = h_2(K_l)$. Let $\mathcal{H}$ denote the set of all possible such sequences, of length n, of hash pairs. Once we have placed the n keys we wish to determine the expected probe length for an unsuccessful search; let $\mathcal{U}$ be the set of $m(m-1)$ possible hash pairs for this new key. We have $|\mathcal{H} \times \mathcal{U}| = \big(m(m-1)\big)^{n+1}$, and each of these elements is equally likely. On the probability space $\mathcal{H} \times \mathcal{U}$ we define the random variable $Y^{\mathrm{DH}}_{\alpha,m}$; for $(h, u) \in \mathcal{H} \times \mathcal{U}$, $Y^{\mathrm{DH}}_{\alpha,m}(h, u)$ is the integer computed as follows: fill a table according to double hashing using the hash pairs specified by h, and then return the number of probes double hashing would use to search unsuccessfully for a key with hash pair u in the resulting configuration. Then the quantity $P^{\mathrm{DH}}_{\alpha,m}(k)$ appearing in the Theorem is simply

$$P^{\mathrm{DH}}_{\alpha,m}(k) = \Pr\{Y^{\mathrm{DH}}_{\alpha,m} \geq k\}.$$

```
procedure UsuallyDoubleHash(n)
begin
  k := 0; /* Number of original keys inserted so far */
  f := 0; /* Number of table positions filled so far */
  while k < n and f < ⌊(1 + 2δ)n⌋ do
    begin
      if there exists an empty slot x for which ξ(x) > (1+δ)/(m−f)
        then VeryUnlikely: exit this while-loop;
        else UsualCase: if flip(1/(1 + δ))
          then begin
                 k := k + 1;
                 /* Note that the probability that slot x is filled by the statement
                    below is ξ(x) */
                 insert K_k into the table according to double hashing, and color it
                    green;
               end
          else begin
                 choose an empty table location x to be filled according to the
                    probability distribution

                        g(x) = δ^{-1}( (1 + δ)/(m − f) − ξ(x) ),

                 and color it red;
               end;
      f := f + 1;
    end; /* of while-loop */
  while f < ⌊(1 + 2δ)n⌋ do
    insert an extra red point into the table according to uniform hashing;
end; /* of procedure */
```

Figure 2
Procedure *UsuallyDoubleHash*

Let $\mathcal{T}$ be a probability space used to determine the values returned by **flip**, and the choices of locations for red points, in UDH. (The symbol $\mathcal{T}$ is intended as a mnemonic for the *tossing* of coins.) The probability space used in the analysis will be the product space $\mathcal{S} = \mathcal{H} \times \mathcal{U} \times \mathcal{T}$. The random variable $Y^{\mathrm{DH}}_{\alpha,m}$ can easily be extended to be defined on this space by simply ignoring the component from $\mathcal{T}$. The variable $Y^{\mathrm{UDH}}_{\alpha,m}$ maps an element $(h, u, t) \in \mathcal{S} = \mathcal{H} \times \mathcal{U} \times \mathcal{T}$ to a value computed in the following way: fill a table according to h and t by the algorithm UDH, and then return the number of probes that double hashing would use to search for a key with hash pair u in the resulting configuration. We define

$$P^{\mathrm{UDH}}_{\alpha,m}(k) = \Pr\{Y^{\mathrm{UDH}}_{\alpha,m} \geq k\}.$$

Figure 3 depicts a simplified possible plot of $\xi(x)$ when for all empty slots x, $\xi(x) < (1 + \delta)/(m - f)$ (the "UsualCase" in the algorithm). The scales on the axes in

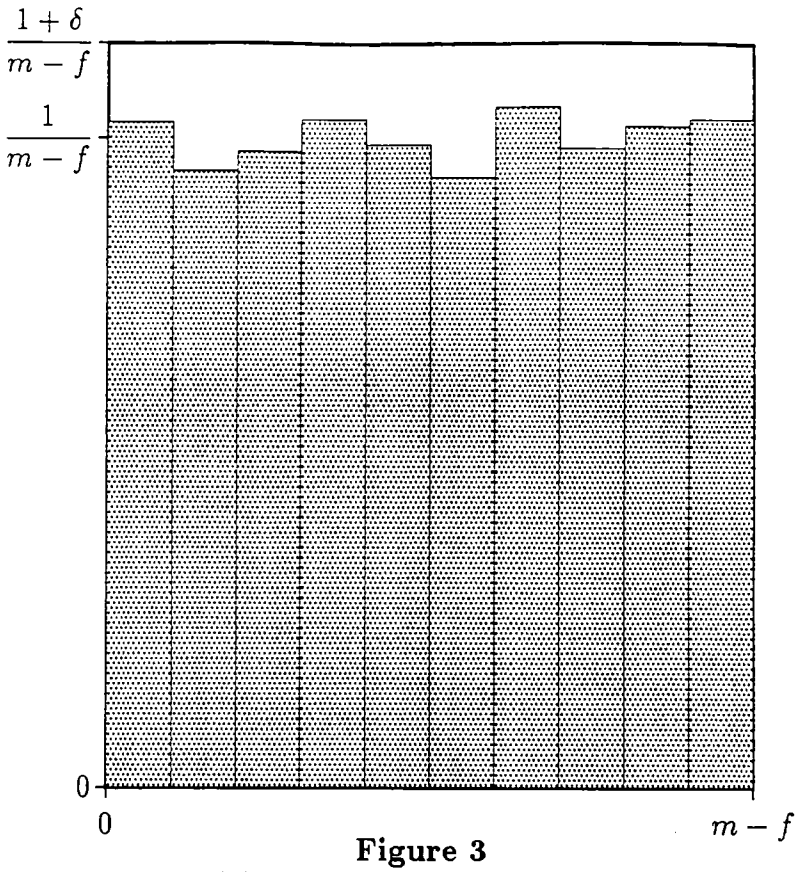


Figure 3
A plot of $\xi(x)$, for the $m - f$ empty table positions.

that figure are not equal: For convenience we have scaled the horizontal axis so that each of the vertical bars has width 1. Note that the shaded region in that figure has area 1, and the white region (between $\xi(x)$ and $(1+\delta)/(m-f)$) has area δ. The probability distribution $g(x)$ used in the algorithm is just the white region, normalized to have area 1.

Note that the block labelled "UsualCase:" can be viewed as drawing a point uniformly from the rectangle shown in Figure 3, coloring it green if it is drawn from the shaded portion of that rectangle and red otherwise. (The procedure **flip** decides first which color to use, and then ξ is used to select a slot for a green point, or g is used to select a slot for a red point.) Hence it is not hard to see that the distribution of table configurations (and slot colors) is unaffected if we replace the block with the following:

> UsualCaseVariant: Choose the next table location x to be filled according to uniform hashing. Choose whether x is filled with a red point or a green point by letting
>
> $$\Pr\{x \text{ is green}\} = \frac{\xi(x)}{(1+\delta)/(m-f)}.$$

We can now begin to make comparisons between the distributions of the number of probes for *UsuallyDoubleHash* (UDH), double hashing (DH), and uniform hashing (UH).

Lemma 4.

$$P^{\mathrm{UDH}}_{\alpha,m}(k) = P^{\mathrm{UH}}_{\lfloor(1+2\delta)\alpha\rfloor,m}(k).$$

Proof. By the UsualCaseVariant described above, if we ignore the colors of inserted points, UDH is simply performing uniform hashing, and inserts precisely $\lfloor(1+2\delta)m\rfloor$ points. (Note that regardless of why we exit the first **while**-loop, the second loop pads the table to contain $\lfloor(1+2\delta)m\rfloor$ points.) Thus the length of a probe sequence for a new hash pair chosen independently is just the same as that for uniform hashing in a table filled to this size. ∎

In order to complete the proof we compare the distributions of the number of probes for DH and UDH. Usually we have $Y^{\mathrm{DH}}_{\alpha,m} \leq Y^{\mathrm{UDH}}_{\alpha,m}$, though this can fail if we did not insert all n of the original keys. Let E_{fail} be this event, i.e., that $k \neq n$ at the end of UDH.

Lemma 5. *If $\omega \in \mathcal{S} - E_{\mathrm{fail}}$, then $Y^{\mathrm{DH}}_{\alpha,m}(\omega) \leq Y^{\mathrm{UDH}}_{\alpha,m}(\omega)$.*

Proof. If all n keys were placed by UDH, then DH and UDH insert the same keys (with the same hash sequences), except that UDH occasionally adds an extra point. Then by the monotonicity of double hashing, the configuration produced by UDH dominates that which would have been produced by DH. ∎

Fortunately, event E_{fail} is quite unlikely.

Lemma 6. $\Pr\{E_{\mathrm{fail}}\}$ *is c-polysmall.*

Proof. First consider the case that the **exit** statement labelled "VeryUnlikely:" was ever executed. By Lemma 3, equation (1.2), and equation (3.1), for each insertion the probability that the next point will be placed by the block labelled "VeryUnlikely:" is c-polysmall. Since the number of insertions is $O(m)$, the probability that this block is ever invoked is c-polysmall.

Next suppose that VeryUnlikely is never invoked. Then it must be that there were $(1+2\delta)n$ calls to **flip** but fewer than n **true**'s among the results. Hence we can bound the probability of this case by the probability that there are fewer than n successes in $(1+2\delta)n$ Bernoulli trials with success probability $1/(1+\delta)$. By Lemma 1 and equation (3.1) this be shown to be c-polysmall in n, and hence also in m, for any fixed $0 < \alpha < 1$. ∎

Proof of Theorem 1, concluded. Combining Lemmas 4, 5, and 6, we immediately obtain the upper bound given in equation (3.2).

We now give a brief sketch of the proof of the lower bound for $P^{\mathrm{DH}}_{\alpha,m}(k)$, which uses the procedure UsuallyDoubleHash′ shown in Figure 4. The sampling procedure required now does not need to insert additional points, but rather makes the distribution probability for filled slots uniform by occasionally rejecting a point; this is quite similar to the resampling used in another context in [KK82]. The result is a table with configurations which are distributed as if filled by uniform hashing and which are dominated by those that would have been obtained if we had used double hashing; moreover, they are very likely to contain at least $(1-\delta)n$ points. We omit the details. ∎

```
procedure UsuallyDoubleHash'(n)
begin
   k := 0; /* Number of original keys inserted so far */
   f := 0; /* Number of table positions filled so far */
   while k < n do
      begin
         if there exists an empty slot x for which ξ(x) < (1-δ)/(m-f)
            then VeryUnlikely: report failure and return;
            else UsualCase:
               begin
                  for each empty slot x do
                        RejectProbability(x) := 1 - (1-δ)/(m-f) / ξ(x);
                  insert K_k into the table according to double hashing, and let x denote
                     the slot which is filled;
                  if flip(RejectProbability(x))
                     then make slot x empty again (discarding key K_k);
                     else f := f + 1;
                  k := k + 1;
               end /* of UsualCase */
      end /* of while-loop */
end; /* of procedure */
```

Figure 4
Procedure *UsuallyDoubleHash'*

CHAPTER 4
Conclusions and Related Work

In [BBS90] the probability distribution of the insertion costs for a type of hashing algorithm known as *random probing with k-ary clustering* is analyzed. In this type of hashing algorithm, we assume that the key distributions and hash functions are such that the first k positions in the probe sequences for successive keys are independent. The remaining positions in probe sequences are a function of the first k; this function is chosen randomly but fixed at the beginning of the run of the algorithm. For $k = 1$, the closed form of the probability distribution is given, and for $k \geq 2$, they show that the probability distributions are asymptotically identical to that of uniform hashing, enabling them to conclude that all moments are asymptotically equal to those of uniform hashing.

The results discussed here were originally presented in [LM88]. There we only presented bounds on the expectation of $Y_{\alpha,m}^{\mathrm{DH}}$, rather than on its distribution. However, [SS90] noted that bounds on the distribution of $Y_{\alpha,m}^{\mathrm{DH}}$ were implicit in the results; using this observation, we have, as [BBS90] did for algorithms with k-ary clustering, written the present paper in a way which explicitly addresses this distribution. In particular, in view of the bound in Theorem 1, we have the following, which answers a question mentioned in [BBS90].

Corollary 2. *For any fixed α and $q \geq 0$ as $m \to \infty$ over the primes,*

$$\mathsf{E}[(Y_{\alpha,m}^{\mathrm{DH}})^q] \sim \mathsf{E}[(Y_{\alpha,m}^{\mathrm{UH}})^q].$$

There are several ways in which these results might be strengthened. One of the most interesting, recently addressed in [SS90], is the question of how one might analyze open addressing schemes if we weaken the assumption that all of the keys to be inserted are independently distributed. For example, by using universal hashing schemes, one might be able to eliminate any assumptions about the distribution of keys and still enforce the condition that any set of $\log n$ hash pairs are independent. This complicates the problem considerably, but Schmidt and Siegel were able to obtain results under such assumptions [SS90].

PART II
Parallel Depth First Search of Planar Directed Graphs

CHAPTER 1
Background Information

Since depth first search has proven to be a very important technique in sequential algorithms for graph problems, it seemed natural to look for a parallel depth first search algorithm. When Reif [Rei85] showed that the problem of finding, for a general graph, the unique depth first search ordering of the vertices specified by a given set of adjacency lists for vertices was P-complete, it may have appeared unlikely that an NC algorithm for depth first search existed. Since then, however, Smith [Smi86] has shown that finding a depth first search ordering of planar undirected graphs was in NC, and Aggarwal and Anderson [AA87] have found a randomized NC algorithm to give a depth first search ordering for the general undirected graph. Later papers brought the processor count used in the planar undirected case to linear [HY88, JK88, Sha88].

For directed graphs, Kao [Kao88] first placed the problem for planar graphs in NC, and Aggarwal, Anderson, and Kao [AAK89] have shown a randomized NC algorithm for the case of general directed graphs. Kao and Klein [KK90] have recently given a depth first search ordering for planar directed graphs using a linear number of processors, but the time bound is $O(\log^8 n)$.

We present an algorithm for the depth first search of a planar directed acyclic graph with k sources using $O(n)$ processors and $O(\log k \log n)$ time on a CRCW PRAM model. (We will use often use the common abbreviation *dag* for directed acyclic graph.) For the case of planar *st*-graphs, which are dags with a single source and a single sink on the same face, we present a simple optimal algorithm which gives the depth first search in $O(\log n)$ time with $O(n/\log n)$ processors on an EREW PRAM, assuming that the embedding is given. We then use the *st*-graph algorithm to construct an algorithm with the same time and processor bounds for the case in which the single source and single sink are not necessarily on the same face. For the case of a single-source multiple-sink planar dag, we have an $O(\log n)$ time $O(n)$ processor EREW algorithm, again assuming the embedding is given. We also give a simplified variant of the depth first search of a multisource planar dag which can be used to solve the single source reachability problem for a planar directed acyclic graph in $O(\log^2 n)$ time and $O(n)$ processors on an CRCW PRAM. Since an $O(\log^4 n)$ algorithm for this problem is used as a subroutine by Kao and Klein in their depth first search for the general planar directed graph, this will lower their time bound by a factor of $\log^2 n$.

The algorithm relies heavily on the Euler tour technique [TV85]. We introduce the idea of specifying a particular depth first search tree for a planar graph, one in which

the Euler tour which accompanies the depth first search defining the tree is also planar and crosses no tree edges. Given a root and an orientation about the tree (clockwise or counterclockwise), this gives a unique depth first search tree. We exploit the properties of this depth first search which we will call the Euler tour depth first search or ET dfs to make the correct local decisions. (ET stands for Euler tour and dfs is a common abbreviation for depth first search.) Using this, we get a simple optimal algorithm for the depth first search of a planar directed acyclic graph with one source and one sink, which may be on separate faces.

For the case of single-source multiple-sink planar dags, we look at the dual of the graph, which will be strongly connected. We find a particular spanning tree of the dual, and find the Euler tour numberings of the dual edges. This in turn will help determine which in-edge is chosen by an ET dfs of the original graph. For the general case of the planar dag, the problem was to find and separate out the vertices reachable from different sources. Again we look at the dual of the graph. In the dual, sources and sinks will appear as cyclic faces. To remove multiple sources, we iteratively find and combine sets of vertices reachable only by a source or set of sources, none of which is the designated root source, and process them out until only the root source is left. Using the information gained from the dual, we are then able to cut the original dag into multiple components, each of which is a single-source multi-sink dag. The simpler algorithm can be run on each of them, and the results are recombined to form an ET dfs forest.

In the remainder of this chapter we will give some of the terminology and basic techniques for planar digraphs that we use in the work. Chapter 2 will discuss the Euler tour depth first search and define Right Hand In-Path and Right Hand Out-Path structures of a planar graph. The properties of these structures are used extensively in finding the depth first search of the graph. In Chapter 3 we give the algorithms for finding the Euler tour depth first search of single source dags. There are three algorithms, increasing in complexity, for the very specialized case of a planar *st*-graph, for the slightly more general case of a planar single-source single-sink dag, and finally for the case of a planar single-source dag. In Chapter 4 we deal with the case of the planar multiple-source dag. Chapter 5 gives a summary and discusses possible future work in this area.

1.1. Definitions and terminology

1.1.1. PRAM: model of parallel computation

In this dissertation, we will be using the Parallel Random Access Machine or PRAM model of parallel computation. A PRAM consists of independent processors, which communicate with each other through a shared global memory. Each processor may also have some private memory. In one time step, each processor can execute a single RAM operation, or read or write into a single memory location.

The PRAM model is subdivided into models depending on whether more than one processor can access a specific memory location in the same time step. In the most restrictive model, the Exclusive Read Exclusive Write or EREW PRAM, no concurrent access to a memory location is allowed. In the Concurrent Read Exclusive Write or CREW PRAM, more than one processor can read from a single memory location, but only one processor can write in a given location during one time step. In the Concurrent

Read Concurrent Write or CRCW PRAM, more than one processor have access to a memory location for both reading and writing.

In the case of the CRCW PRAM, the ways of resolving write conflicts lead to three commonly used CRCW PRAM models. In a COMMON CRCW, concurrent writes are allowed only when all processors writing to a given location are writing the same value. If any one processor among those attempting to write can succeed, where the choice of processor is arbitrary, the model is called the ARBITRARY CRCW. If there is a priority ordering among the processors and the highest priority processor among those attempting to write always succeeds, it is a PRIORITY CRCW PRAM.

Although there are many PRAM models, the most restrictive EREW PRAM can simulate one step of the most powerful n processor PRIORITY CRCW PRAM model in $O(\log n)$ steps using the same number n of processors. (For a survey of these and other results relating to the PRAM model, see [KR88].)

1.1.2. Graph terminology

We use the standard terminology for directed graphs [AHU74]. A graph G consists of a set V of vertices $\{v_1, v_2, \ldots v_n\}$ and a set E of directed edges $\{e_1, e_2, \ldots e_m\}$. An edge $e = (u, v)$ is an edge directed from vertex u to vertex v. The vertex u is called the *tail* of edge e and v is called the *head* of e. The edge e is an *in-edge* for the vertex v, and an *out-edge* for the vertex u. A vertex with no in-edges is called a *source*, and a vertex with no out-edges is called a *sink*. A *directed path* from vertex u to vertex v is a set of edges $\{f_1, f_2, \ldots f_k\}$ such that the tail of f_1 is u, the head of f_k is v, and the head of f_i is the tail of f_{i+1} for $i = 1$ to $k - 1$. Any vertex which is a head or a tail of an edge in a directed path will be said to be a vertex *on the path*. In a *simple* directed path, a vertex on the path can be the head of only one edge and the tail of only one edge in the path. A directed path from vertex u back to u is called a *cycle*; a *simple cycle* is a simple directed path from a vertex u back to u. A directed graph with no cycles is called *acyclic*.

In this dissertation we assume that we are given a directed acyclic planar graph G with a source vertex and one of its out-edges designated to be the start of the depth first search of G. The designated source will be called the *root* source. We will assume that the combinatorial embedding of the planar graph G is given. The embedding is specified by giving all the edges incident on a vertex in the graph in cyclic order around the vertex. For convenience we will define the cyclic ordering to be clockwise about the vertex. If the embedding is not given, it can be obtained using Ramachandran and Reif's planarity algorithm using $O(\log n)$ time and at most $O(n)$ processors in the CRCW PRAM model [RR89]. We note that this specification gives the embedding of G on the surface of a sphere.

We will also assume that the graph is connected. If necessary, an optimal CRCW algorithm for finding connected components for undirected planar graphs can be run [Hag88]. Without loss of generality, we assume no multi-edges or self-loops in G; a depth first search of a graph with multi-edges and self-loops removed will give a valid depth first search of the original graph with multi-edges and self-loops.

In this work, we make use of the constraints that an embedding of a planar graph provides. In order to specify the relationship between parts of a graph given by the

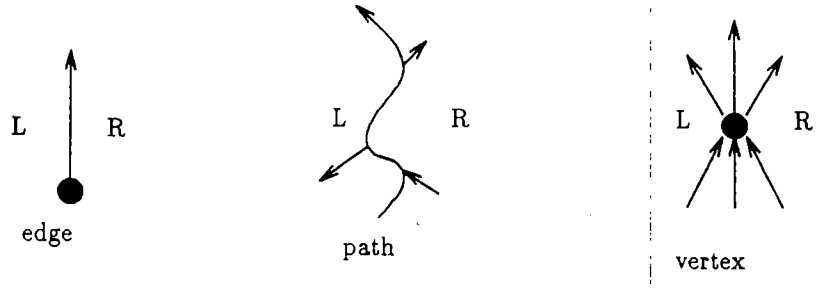


Figure 1
Right and left orientation

embedding, we give the following definitions for use in this dissertation. We will define the *right* side of an edge to be the side to the right when we travel in the direction the edge is pointing. The *left* side is the opposite side of the edge. The definitions can be expanded for the right and left sides of a directed path, as well as for a group of consecutive in-edges or a group of consecutive out-edges at a vertex which is neither a source nor a sink. Thus the rightmost in-edge of a group of in-edges is the first in-edge in the group encountered while going clockwise about the vertex. The rightmost out-edge of a group of out-edges will be the last out-edge in the group encountered while going clockwise. At a source or a sink, we cannot distinguish any rightmost or leftmost edge. However, for a source, we are usually given as part of the problem definition, the first out-edge on the adjacency list of the source. This distinguished out-edge can serve the same function as the rightmost out-edge does for non-source non-sink vertices. (Figure 1)

We will make use of the concept of a simply-connected region from differential geometry [BB78, p. 503]. For a plane or for the surface of a sphere, a simply-connected region will have no holes in it, so that its boundary will be a single simple cycle. To prevent confusion since we will be using "connected" in terms of graphs, we will call a simply-connected region a *hole-free* region.

In some sections we will specify that a certain face be the outside face. This is done to aid in description and in proving correctness, and is not an integral part of the problem definition or the depth first search algorithms.

Once an outside face has been specified, we can define two directions for a simple cycle in the embedded graph. One hole-free region of the sphere lies to right of the path creating the cycle, and one region to the left. If the outside face is in the region to the left, the simple cycle will be called *clockwise*; if the outside face is in the region to the right, the cycle is *counterclockwise*. In both cases, the region of the sphere which does *not* contain the outside face will be said to be the *inside* of the cycle. (Figure 2)

Once an embedding is given, we can define the *crossing* of two paths. We will say that Q *crosses* P *from the right*, or equivalently, that P *crosses* Q *from the left*, if there exists a vertex v which is on both paths and is not an end vertex of either path and if the in-edge to v on path Q is to the right of path P and the out-edge from v on path Q is to the left of P. Two paths P and Q cross if path P crosses path Q from either the right or the left.

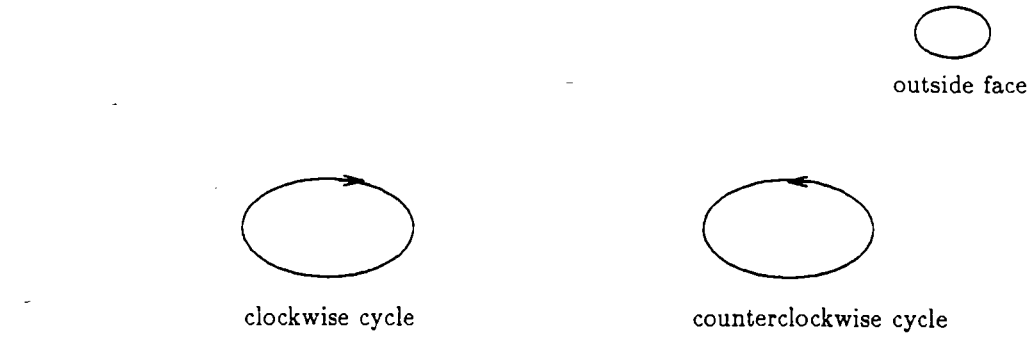


Figure 2
Cycle orientation

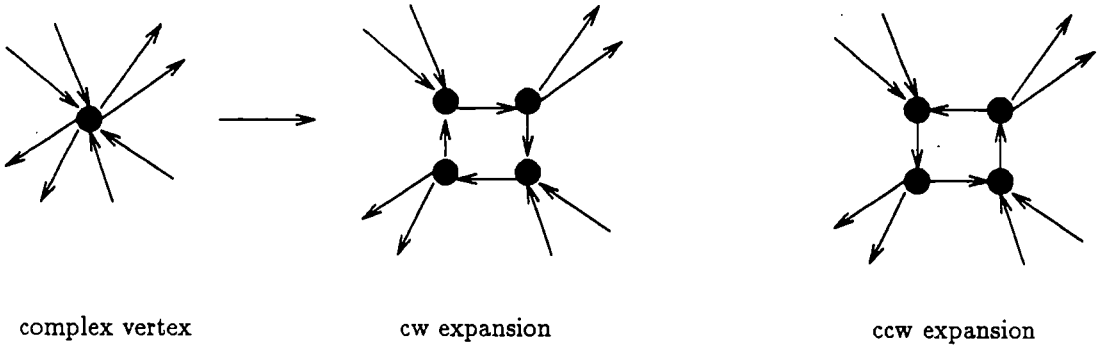


Figure 3
Vertex expansion of a complex vertex

A face will also be associated with a list of edges in cyclic order which make up the boundary of each face; these lists can be constructed from the edge lists for the vertices. If the edges about the face are all oriented in the same direction, we will call the face a *cycle face*. If the cycle is clockwise, it is called a *positive face*; if it is counterclockwise, it is a *negative face*. These definitions of positive and negative faces, as well as the following definitions of vertex expansion and contraction, are either the same or similar to those used by Kao and Shannon [KS90].

A vertex will be called *simple* if all the in-edges are in one consecutive group in the cyclic order of edges about the vertex. This includes the cases when there exist no in-edges and when all edges are in-edges. Otherwise, the vertex will be called *complex*. Given a complex vertex v with groups of in-edges $A_1, A_2, \ldots A_k$ and groups of out-edges $B_1, B_2, \ldots B_k$ alternating clockwise $A_1, B_1, A_2, B_2, \ldots A_k, B_k$ about v, we will call the following operation *cw vertex expansion*: we create $2k$ copies of vertex v connected by a clockwise cycle of edges. If we traverse along the cycle, the groups of edges A_i's and B_i's will be connected in the same clockwise order as they were to v, with A_i and B_i connected to vertex copies v_{2i} and v_{2i+1}. If the vertex copies are linked by a counterclockwise cycle, we will call it a *ccw vertex expansion*. The result of either vertex expansion is to take a complex vertex and replace it with $2k$ simple vertices. (Figure 3)

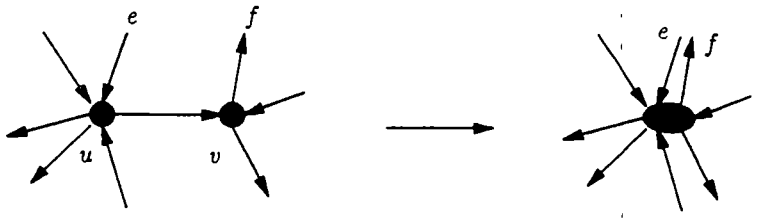


Figure 4
Vertex contraction along an edge

Besides expanding a vertex, we can contract two vertices into one by contracting an edge between them. In a graph with the combinatorial embedding given, if u and v are the vertices to be contracted into one along the edge (u, v), the embedding may be retained by cutting the cyclic ordered lists of edges about u and about v at the edge (u, v) and splicing them together so that the relative cyclic ordering is preserved. Thus if edge e was just before (u, v) in the cyclic list for u, it would now be just before f, where f would be the edge just after (u, v) in the cyclic list for v. We note that such contractions may produce multi-edges and self-loops; in the context in which we use vertex contraction, these will not prove to be a problem. We can also expand this idea of contraction for the case of two vertices which are on the same face. Suppose we create a new edge between two such vertices bisecting the face. This would not violate planarity. We then contract along the just created edge in the way just described.

In Chapter 4, we will present algorithms in which all vertices lying in a simply-connected or hole-free region are replaced by a single vertex. This is equivalent to contracting all vertices within the hole-free region and removing all self-loops. We will say that the new vertex *inherits* the edges of the removed vertices and their embedding to indicate that the list of all edges crossing the boundary of the hole-free region ordered clockwise along the boundary will be the clockwise list of edges for the new vertex.

We also make extensive use of the properties of the dual of a graph. The following is some standard terminology for dual graphs, except that we have fixed the relative directions of the edge and dual edge. We define the *dual* of an embedded graph G as a graph G^* such that the vertices of G^* correspond to the faces of G, and there exists an edge from vertex x^* to vertex y^* in G^* if the corresponding faces F_x and F_y in G share an edge which is going in a counterclockwise direction around the boundary of F_x. The cyclic ordering of the edges for a vertex in the dual G^* will be the same as the cyclic ordering of the edges on the boundary of the corresponding face in the graph G. This specification for the embedding of the dual allows us to consider both G and G^* to be embedded on the same sphere with the vertex of the dual placed inside the associated face of G, so that G^* is the geometric dual of G. (Figure 5)

We will for convenience say that the dual edge e^* in G^* crosses the edge e in G from the left. This is not the same as the definition of crossing paths, given previously. Then the two paths were part of the same graph; here the two edges belong to a graph and its dual respectively. We are making use of the joint embedding of the two graphs, as noted above. In that joint embedding, we see that a positive face of G^* encloses a source (a

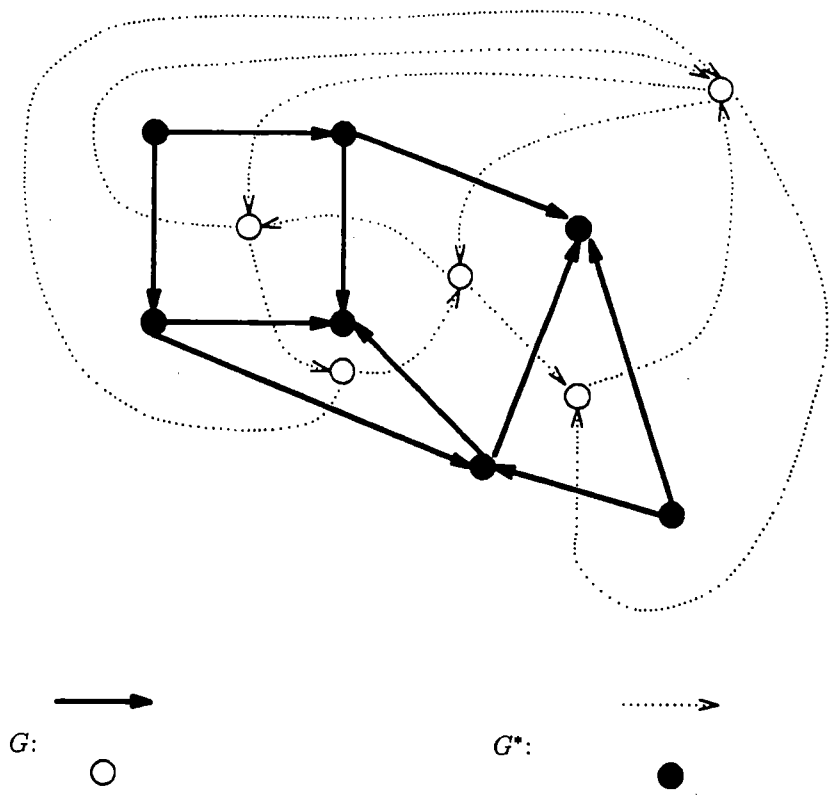


Figure 5
A graph G and its dual G^*

vertex with only out-edges) in G, and a negative face of G^* encloses a sink (a vertex with only in-edges) in G. A cw expansion on a vertex x^* in G^*, corresponds to the creation of a new source in G in face F_x. A ccw expansion of a vertex in G^* corresponds to the creation of a new sink in G. A vertex contraction in G^* along an edge e^* corresponds to the removal of edge e in G.

We note that a planar graph with n vertices and no multi-edges or self-loops has $O(n)$ faces and $O(n)$ edges. Thus the size of its dual graph will also be $O(n)$. Given a graph G with its embedding, we can find its geometric dual graph G^* using $O(n)$ processors in $O(\log n)$ time in the EREW PRAM model. We conclude by listing the following useful properties of the dual of a planar graph.

Property 1.1 [Har72, p. 113] *The geometric dual of an embedded planar graph is an embedded planar graph with self-loops and multi-edges possible.*

Property 1.2 [KS90] *The dual of a planar directed acyclic graph is a planar strongly connected graph.*

1.2. Basic Techniques

We list some techniques which we use as basic tools in the depth first search algorithms given in this dissertation. A good review of these and some of the other

parallel algorithms mentioned is in [KR88]. One particular technique, the Euler tour technique, is very important to this work and will be treated in the next chapter.

1.2.1. List ranking

The problem of list ranking is as follows: given a linked list, find for each element in the list the number of its predecessors on the list. Represented in terms of a graph, we have a directed graph of n vertices connected in a line with all edges directed away from the head vertex. We wish to label each vertex by its distance from the head vertex. Given n processors, one for each vertex, there is a standard pointer doubling algorithm which will find the list ranking in $O(\log n)$ time on an EREW PRAM [Wyl81]. Solving it optimally with $n/\log n$ processors and $O(\log n)$ time is more difficult, but has been done using deterministic coin tossing technique as an aid in assigning the $n/\log n$ processors among the n vertices. [CV88, AM88]. Using this technique, we can optimally convert any linked list into an array.

1.2.2. Prefix sum

Given an array A of n elements, the prefix sum problem is that of finding all the partial sums:

$$S_i = \sum_{k=1}^{i} A[k]$$

Again there is a simple sequential algorithm which runs in linear time. Using pointer doubling techniques as in list ranking, we can get a parallel algorithm using n processors in $O(\log n)$ time. To convert this to an optimal parallel algorithm using $O(\log n)$ time and $O(n/\log n)$ processors on an EREW PRAM, we can assign each processor to sum up a block of $\log n$ elements in $O(\log n)$ time. Now the smaller prefix sum problem can be solved using the $n/\log n$ processors, and each processor again processes its block of $\log n$ elements to solve the full problem. [LF80]

The same algorithm can be used to find the maximums or minimums instead of sums. By using $O(n/\log n)$ processors and $O(\log n)$ time we can find the maximum $A[k]$ in the range $k = 1$ to i for all i from 1 to n. In this work, we will call this *the prefix maximum* problem.

1.2.3. Operations on trees and circular lists

Suppose we are given a tree with n nodes, in which a node points to its parent. By using pointer doubling, we can calculate $S_i = \sum A[k]$ where $A[k]$ is a value stored at node k and where the summation is over all the nodes on the path in the tree from node i to the root of the tree. This would require n processors and $O(\log n)$ time on a CREW PRAM, since concurrent reads are required. Again, the same technique can be used to find maximums and minimums instead of sums.

Pointer doubling can also be used on circular lists with n elements to find the maximum number $A[k]$ in the list. This can be done with n processors in $O(\log n)$ time in an EREW model.

1.2.4. Graph algorithms for planar graphs

Optimal CRCW algorithms are known for finding the connected components and for finding a spanning tree for undirected graphs [Hag88]. The algorithms run in $O(n/\log n)$ time using $O(n)$ processors. We make use of these algorithms, treating the directed graph as undirected by ignoring the edge direction.

CHAPTER 2
Planar Euler tour depth first search

2.1. Euler tour for a rooted tree

Introduced by Tarjan and Vishkin [TV85], the Euler tour technique for processing trees in parallel is used as a basis for other parallel algorithms. Assume we are given a rooted tree in the form of adjacency lists of edges to children for interior vertices. For each edge (p, c) between a parent node p and a child node c, we create two directed edges, a $downedge(p, c)$ pointing from the parent to the child and an $upedge(c, p)$ pointing from the child to the parent. We obtain an Euler tour traversing these edges by specifying as follows for every edge e_1, which edge e_2 comes next in the traversal. After every $downedge(p, c)$, we traverse $downedge(c, g)$, where (c, g) is the first edge on the adjacency list for c. If c has no children, $downedge(p, c)$ is followed by $upedge(c, p)$. After every $upedge(c, p)$, we traverse $downedge(p, c')$, where (p, c') is next after (p, c) in the adjacency list for p. If (p, c) was the last edge on p's adjacency list, $upedge(c, p)$ is followed by $upedge(p, q)$, where q is the parent of p. In this Euler tour traversal, the down-edges are visited in the same relative order as depth first search visits the associated tree edges. The path taken by this traversal, which we will call the Euler path, can be represented by an ordered list of edges. Since every edge can find the next edge on the list using the above rules in $O(1)$ time, the algorithm for creating this Euler path runs in $O(1)$ time using $O(n)$ processors.

Given the Euler path, we can use a variation of the parallel list ranking algorithm and the prefix sum algorithm to assign preorder, inorder, or postorder numbers to all vertices in the tree. These numbers in turn may be used in many algorithms, for example, determining if two vertices are in an ancestor-descendent relationship in the tree. If the vertices are assigned weights, by using the prefix sum algorithm and the Euler path, we can calculate the weight for all subtrees where the weight of a subtree is the sum of all the weights of the vertices in the subtree [TV85].

In our procedures and algorithms, we repeatedly do two tree trimming operations. Suppose we are given a tree T with some of its edges marked. In an exclusive trimming of the tree, we find the largest subtree with the same root as T such that all its edges are marked. This can be easily done using the Euler tour technique by splicing out any unmarked edges from the tour. In an inclusive trimming, we find the smallest subtree with the same root as T which includes all marked edges. After an inclusive trim, the last edge to any leaf of the trimmed tree will be marked. This can again be implemented using the Euler tour technique. Suppose we assign a weight to the marked edges. Then by using a variation of the subtree weights algorithm mentioned above, we can find the subtrees which contain no marked edges and splice them out of the tree. Thus both these operations have $O(\log n)$ time $O(n/\log n)$ processor EREW PRAM algorithms by using the Euler tour technique.

2.2. Planar Euler tours and depth first search

We now extend the same idea of Euler tour traversal to directed graphs in which every vertex is reachable from a specified root vertex. Suppose we replace each directed edge (u, v) by two directed edges connecting vertices u and v, *downedge*(u, v) pointing in the same direction as (u, v) and *upedge*(v, u) pointing in the opposite direction. We tie the Euler tour traversal of these *downedges* and *upedges* to a sequential depth first search of the original graph G as follows. Let T be the depth first search tree of G with the root vertex as the root of T. If (u, v) is an edge in T, then *downedge*(u, v) is followed by *downedge*(v, w) where (v, w) is the first out-edge from v examined by the sequential depth first search to see if w has been visited previously. If (u, v) is not an edge in T, then *downedge*(u, v) is followed by *upedge*(v, u). After *upedge*(v, u), we traverse *downedge*(u, v') where (u, v') is the out-edge from u next examined by the sequential depth first search after out-edge (u, v). If (u, v) was the last out-edge from u examined by the sequential depth first search, *upedge*(v, u) is followed by *upedge*(u, x) where (x, u) is an edge in T.

This Euler tour traversal will visit *downedges* in the same relative order that the sequential depth first search examines the associated graph edges. If not all vertices in the graph are reachable from a given root vertex, the edge traversal as defined above associated with a sequential depth first search will produce one path for each depth first search tree in the forest, each path forming an Euler tour traversal of the edges associated with those examined by the depth first search tree.

We now further specify this Euler tour traversal for the case of planar graphs. We first specify the embedding of the *downedges* and *upedges* as follows: if in the clockwise ordering of the edges incident on vertex u, each edge (u, v) is replaced by *upedge*(u, v) and *downedge*(v, u) in clockwise order, it will be called a *right-hand* Euler tour. If (u, v) is replaced by *downedge*(u, v) and *upedge*(v, u) in clockwise order, it will be called a *left-hand* Euler tour. An Euler tour traversal will be called *planar* if its path never crosses itself.

If the sequential depth first search of a planar embedded directed graph always examines the out-edges of a vertex in counterclockwise order starting at the in-edge by which the vertex is first visited, the associated right-hand Euler tour traversals as specified above, will be planar. We define such a sequential depth first search to be an *right-hand Euler tour depth first search* or just *Euler tour depth first search*. We call a parallel algorithm an *Euler tour depth first search* (or ET dfs, as noted previously) if it finds the same depth first search forest as that found by a sequential Euler tour depth first search. In this dissertation we present parallel Euler tour depth first search algorithms.

If the out-edges of a vertex are examined in clockwise order by a depth first search, the associated left-hand Euler tour traversals will again be planar. Such a search will be called a *left-hand Euler tour depth first search*. The depth first search forest for this case could also be computed by slight modifications to the parallel ET dfs algorithms given. (Figure 6)

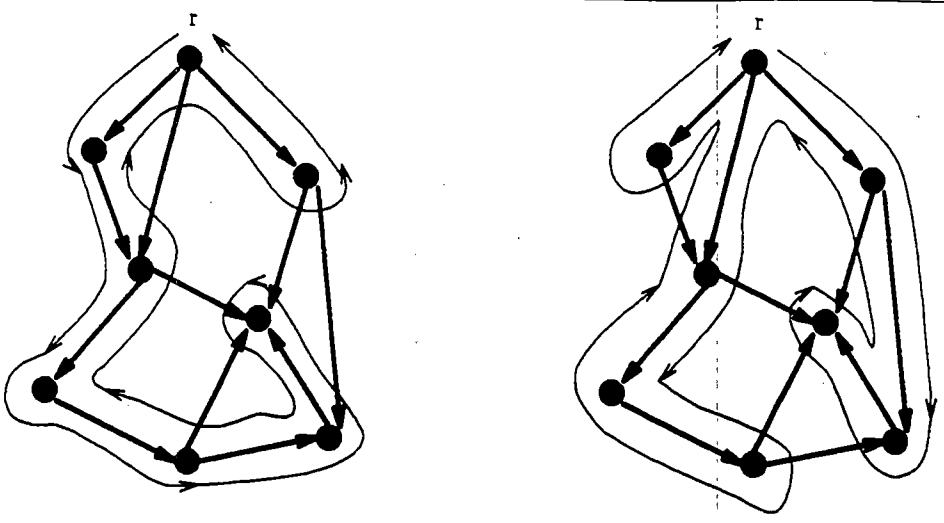


Figure 6
Planar Euler tour depth first search

2.3. RHOP and RHIP structures and the incidence graph of G

Suppose we are given a planar directed graph G, not necessarily acyclic, and its combinatorial embedding. The *incidence* graph G^i of G is defined as follows:

- for every vertex v in G, there is a vertex v^i in G^i,
- for every edge e in G, there is a vertex e^i in G^i, and
- for every edge $e = (u, v)$ in G, there are two edges (u^i, e^i) and (e^i, v^i) in G^i.

The combinatorial embedding of G^i is the same as for G: the cyclic order of edges e_j's for a vertex u in G gives the cyclic order for the corresponding edges (u^i, e^i_j)'s and (e^i_j, u^i)'s. Suppose we define the time a depth first search first visits an edge to be the time the depth first search algorithm examines the edge when it is on the adjacency list of out-edges for a given vertex to see whether or not the head of the edge has already been visited and marked by the search algorithm. Then given an ET dfs of G, an ET dfs of G^i starting at the vertex and edge corresponding to the starting vertex and edge in G, will visit vertices v^i and e^i in the same relative order that the ET dfs of G visits corresponding vertices v and edges e.

Suppose G, and hence G^i, have only simple vertices. We can then define on G^i a subgraph called a *Right Hand In-Path* structure (RHIP) which has all the vertices v^i and e^i of G^i, all edges of the form (e^i, v^i), and those edges (u^i, e^i) in which the corresponding edge $e = (u, v)$ is the rightmost out-edge for u in G. In addition, if s is a source in G and $h = (s, v)$ is the designated first out-edge from s, the edge (s^i, h^i) is chosen to be in the RHIP.

Similarly we define *Right Hand Out-Path* structure (RHOP) in which all vertices of the type v^i in G^i retain their out-edges, but have only one in-edge, the rightmost in-edge for v^i. The corresponding in-edge for v will, of course, be the rightmost in-edge for v.

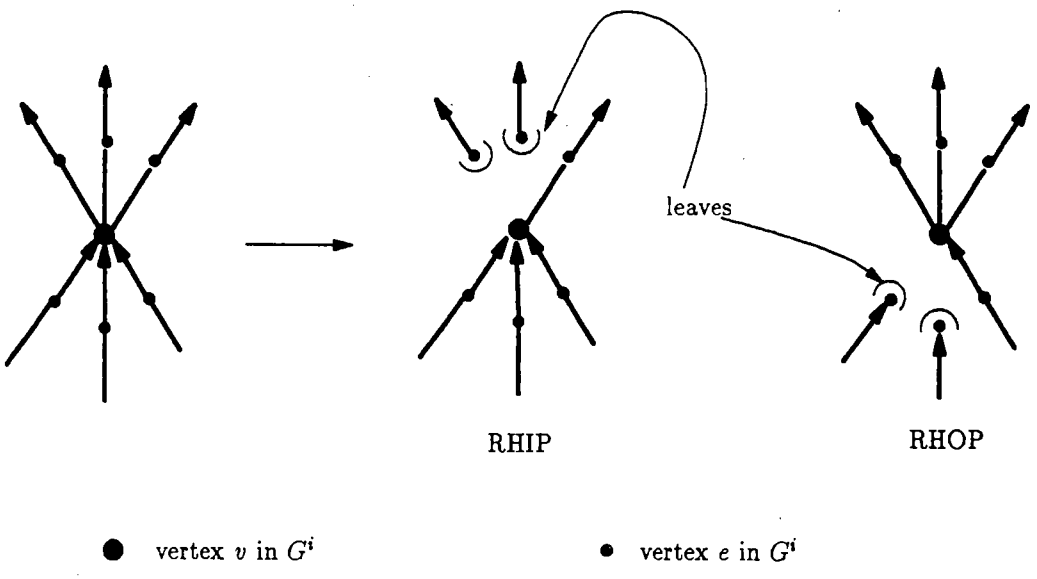


Figure 7
RHIP and RHOP for G^i

The vertices which are sinks (no out-edges) choose no in-edge. This provides a slight asymmetry between the RHOP and the RHIP structures; in the RHOP structure, there exist vertices, the sinks, which are not connected to any other vertices. (Figure 7)

We now give the fundamental property of the RHOP and RHIP structures.

Property 2.1 *On a directed path in an RHOP structure, there are no in-edges incident on the path from the right. On an RHIP path, there are no out-edges incident on the path to the right.*

Since all vertices have out-degree one, the RHIP defines a tree-like structure. If the original graph G (and hence G^i) was acyclic, the RHIP does define in-trees with sinks as roots. If there are multiple sinks, the graph will give an RHIP structure which is a forest with every sink a root of an in-tree. (This is known as a pseudo-forest in [GPS87].) If the original graph was strongly connected, the paths end in positive faces instead of sinks, and we will call each connected component of the *RHIP* structure a *c-tree* to emphasize its tree-like nature. A positive face will be called a *root cycle*, and the separate components which result if all edges in the root cycle were removed are special trees which we will call the *c-subtrees of the c-tree*. The root of a c-subtree will be called the *c-root*. All c-roots are vertices on the root cycle by definition.

For the RHOP case, if the original graph was acyclic, we get a forest of out-trees with sources as the roots of the out-trees. If the original graph was strongly connected, the paths all originate in positive faces, and again we will call each connected component a c-tree, and the positive face a root cycle. (Figure 8)

Although we have defined the RHOP and RHIP structures on the incidence graph of G, it is clear that we could define it on G itself, by having every vertex except the source and sink choosing either the rightmost in-edge or the rightmost out-edge. In the case of

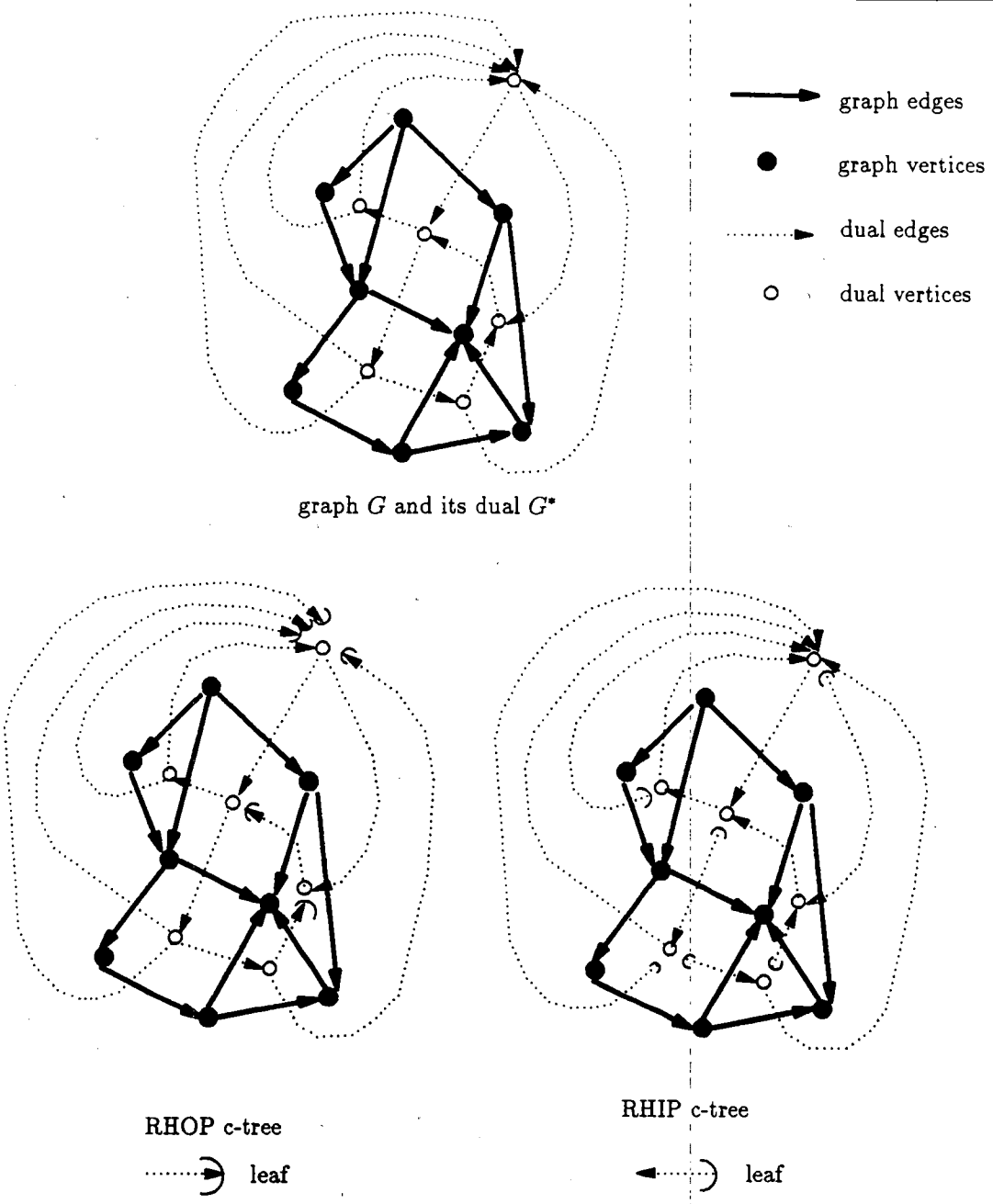


Figure 8
An dag G and the RHIP and RHOP structures of of its dual G^*

the RHIP, the source can choose the designated first out-edge. In the case of graphs with no vertices with degree greater than three (implying all vertices are simple), the RHOP and RHIP structures are almost identical to the F_d and F_c forests of Kao and Shannon [KS90].

In what follows, unless stated specifically otherwise, when we refer to the RHOP and RHIP structures of G we will be referring to the structures defined on G itself, but when we refer to the RHOP and RHIP structures on the dual graph G^*, we will be technically manipulating the incidence graph of G^*. Because of the close relationship between any graph and its incidence graph, the results of operating on the incidence graph translate in an obvious way to the graph. We note that the size of the incidence graph of a planar graph of size $O(n)$ is $O(n)$, and that given $O(n/\log n)$ processors and $O(\log n)$ time, we can create the incidence graph, mark the edges belonging to the RHIP (or RHOP) structure and, using the Euler tour technique, separate out each tree or c-tree component and find its root vertex or root cycle.

2.4. LHOP and LHIP structures

We can also define *Left Hand In-Path* (LHIP) and *Left Hand Out-Path* (LHOP) structures by replacing rightmost by leftmost in the definitions of RHOP and RHIP. By symmetry any properties which we find for the RHOP and RHIP structures can be appropriately altered and attributed to the LHOP and LHIP structures.

We will for the most part only use the RHOP and RHIP structures. However, there is one useful property of the LHOP (and LHIP) structure which we will use.

Property 2.2 *Let a subgraph G be a subgraph induced by the edges of a set of clockwise cycles. Let R be the union of all regions inside the cycles. Then the root cycles of the LHOP c-trees of G will be the boundaries of the region R.*

CHAPTER 3
Depth first search for single source planar dags

3.1. ET dfs of a planar *st*-graph

A planar *st*-graph is a planar acyclic directed graph with one source and one sink, both on the same face. We first present an algorithm for the depth first search of this special graph and later show that it can be readily generalized to the case in which the source and the sink are not on the same face.

We first need the following lemma.

Lemma 3.1 *Given an embedded planar dag with a single source and a single sink, all vertices are simple.*

Proof. (Figure 9) Suppose for a contradiction that we had a complex vertex v. Then there must exist two in-edges of v, which are separated from each other by two sets of out-edges in the cyclic list of edges around v. The two such in-edges must have paths to them from the single source. If we follow the two paths backward from v toward the source, they must meet at some vertex u, which may be the source. If we look at the simple cycle formed by following one of the paths backward from v to u and then forward along the other path from u to v, it will separate the graph into two regions, one inside and one outside the cycle. The two sets of out-edges at v lie in separate regions, implying either the presence of at least two sinks, one in each region, or the presence of a directed path from v which crosses one of the paths from u to v, creating a cycle in the graph from v back to v. Either case contradicts our assumptions; hence there cannot be a complex vertex v. ∎

The following is the basis for Algorithm 1 for the ET dfs of a planar *st*-graph.

Theorem 3.2 *An ET dfs of an embedded planar st-graph will always choose the rightmost in-edge to a non-sink vertex.*

Proof. (Figure 10) Suppose an in-edge to vertex v was chosen which was not the rightmost in-edge. Then the vertex w from which the rightmost in-edge is coming must be visited after v is visited. Moreover, since this an acyclic graph, w cannot be a descendent of v, and the dfs must visit w after visiting all the descendents of v. There exists a path from v to the sink, all the vertices of which must have been visited before the ET dfs traversal returns to v. Since all vertices on the path from the source to v must also have been visited, this creates a path from the source to the sink, all vertices of which have been visited at the time the ET dfs traversal finishes with v. At this point the Euler tour must be on the left side of the path while w is on the right side of the path. In order to visit w, the Euler tour must either cross over itself, or follow some path which will enclose either the source or the sink but not both. The first case contradicts the planarity condition, while the second is impossible if both source and sink are on the same face. Thus there cannot have been such a vertex w. ∎

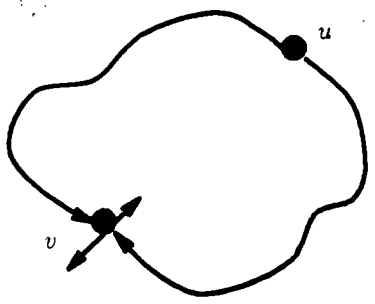


Figure 9
Illustration for lemma 3.1

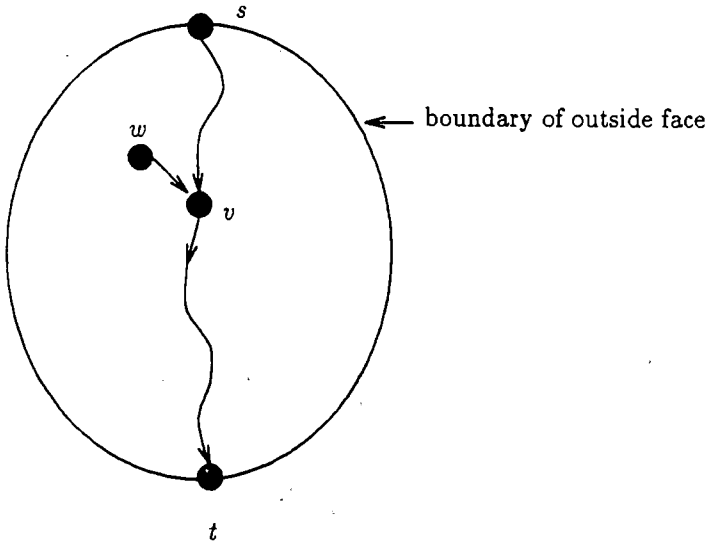


Figure 10
Illustration for Theorem 3.2

Because of Theorem 3.2, the RHOP structure of G gives the ET dfs tree of an st-graph. The only complication is that the sink is not connected in an RHOP structure; however, the in-edge for the sink can be easily found after the rest of the ET dfs tree has been found.

Algorithm 1. ET depth first search for a planar st-graph

1. Find the RHOP structure of the graph. This is possible since all vertices are simple.
2. Using the Euler tour technique, find the preorder numbering for all vertices in the tree defined in step 1.
3. Among all vertices with edges to the sink, find the vertex with the lowest preorder number. Add the sink and the edge from the chosen vertex to the sink, to the tree defined in step 1.
4. Redo the Euler tour technique to find the preorder numbering of the ET depth first search tree including the sink.

The first step can be done in constant time by $O(n)$ processors, or in $O(\log n)$ time by $O(n/\log n)$ processors in the EREW model. All other steps can be done in $O(\log n)$ time with $O(n/\log n)$ processors, so Algorithm 1 is optimal.

3.2. ET dfs of a single-source single-sink planar dag

For a single-source single-sink planar dag whose source and sink are not on the same face, we use the following technique to convert it into a st-graph without modifying the depth first search tree.

We note that for any vertex, the path defined by RHIP gives the direction that the ET dfs would take from that vertex for as long as it meets no previously visited vertex. For the case of single source and single sink graph G, the RHIP on G defines a single tree with the source as one of its leaves. The path on this tree from the source to the sink must therefore be the first set of edges followed by the dfs. Using this we get the following algorithm. All steps can be accomplished using $O(n/\log n)$ processors in $O(\log n)$ time on an EREW PRAM.

Algorithm 2. ET dfs for single-source single-sink planar dag

1. Find the RHIP structure of the graph.
2. Find the path P from the source to the sink on the RHIP tree.
3. For all vertices on the path P, take all in-edges which are incident on the path from the right and make them point to the sink. Now the source and the sink are both on the same face. Note that by Property 2.1 there are no out-edges incident on the path on the right.
4. Run the algorithm for ET dfs for a planar st-graph.

We note that since the path found in step 2 is the first path taken by the dfs, the edges shifted in step 3 could not have been in the dfs tree. Thus they can be shifted to point to the sink at the end of the path without changing the dfs tree of G.

3.3. ET dfs of a single-source multiple-sink planar dag

For the case of a single-source multiple-sink planar dag G, we can find the ET dfs by using the RHOP structure of the dual graph G^*. Algorithm 3 is a brief summary of the procedure.

Algorithm 3. Depth first search of a single-source multiple-sink planar dag G

1. Find the dual graph G^*.
2. Do a ccw expansion of the complex vertices of G^*. This is equivalent to adding new sinks to G. They can be easily removed at the end.
3. Find the RHOP structure of G^*. This will be a c-tree whose root cycle will form the boundary of the positive dual face associated with the source of G.
4. Let h be the first edge followed out of the source by the dfs in G. Let h^* be the corresponding edge in G^*. Then h^* will be an edge on the positive root cycle.

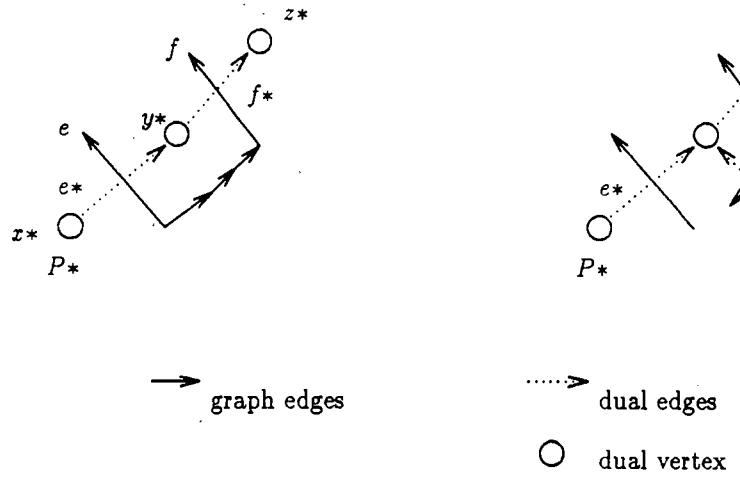


Figure 11
Illustration for lemma 3.3

Cut off h^* from the dual vertex t^* to which it points, thus cutting the cycle and creating an actual tree out of the RHOP c-tree structure.

5. By the previous step, t^* is the root of the RHOP tree. Do an ET dfs of the out-tree starting at t^* and following as the first path, the edges which formed the root cycle until it reaches h^*. For every edge in the RHOP tree, including the cut-off leaf edges, find the postorder numbering using the ET dfs.
6. For every vertex in G, choose as its in-edge for the ET dfs tree the edge whose corresponding dual edge has the lowest postorder number.
7. Use the ET technique to find the preorder numbering of the ET dfs tree found in the previous step.

In order to prove that the algorithm finds the correct in-edge for every vertex of G, we first prove the following lemma and a corollary, which states a useful property of an RHOP path.

Lemma 3.3 *Let P^* be a path of an RHOP structure of a dual graph G^*. Let x^*, y^*, and z^* be three adjacent vertices on P^* such that e^* goes from x^* to y^*, and f^* goes from y^* to z^*. Let e and f be edges of graph G which cross e^* and f^* respectively from the right to the left. Let F_y be the face in G corresponding to y^* in G^* so that e and f are both on the boundary of F_y. Then if there exist any edges on the boundary of F_y between the tails of e and f on the right side of P^*, they are all oriented counterclockwise around F_y.*

Proof. (Figure 11) Suppose for a contradiction that there were some edge g oriented clockwise on the boundary of F_y between e and f. Then there would exist a dual in-edge g^* to y^* which the RHOP path would choose before e^*. Since the RHOP path chose e^*, we have a contradiction. ∎

Corollary 3.4 *If an edge e of G crosses an RHOP path P^* of the dual of G, then there exists a directed path P (possibly degenerate) in G which starts at the source which is enclosed by the root cycle of the RHOP c-tree and ends at the tail of e. Moreover, the path P will parallel the RHOP path and lie directly to the right of P^*. (More specifically,*

P is composed of the boundary edges for the faces of G whose associated dual vertices are on P^*. They lie to the right of P^* and have a counterclockwise direction about their respective faces.) All edges in G incident on the path P on the left side will be out-edges crossing the RHOP path. ∎

A very similar lemma and corollary can be proven for the RHIP path structures. We state the corollary.

Corollary 3.5 *If an edge e of G crosses an RHIP path P^* of the dual of G, then there exists a directed path P (possibly degenerate) in G to the tail of e from the source which is enclosed by the root cycle of the RHIP path. Moreover, the path P will parallel the RHIP path going in the opposite direction, and lie directly to the left of P^*. (More specifically, P is composed of the boundary edges for the faces of G whose associated dual vertices are on P^*. They lie to the left of P^* and have a clockwise direction about their respective faces.) All edges in G incident on the path P on the right side will be out-edges crossing the RHIP path.* ∎

We now show that Algorithm 3 does find the ET dfs of a planar single source dag.

Theorem 3.6 *Let e and f be two in-edges to a vertex v in a single source planar dag G. Assume e and f are adjacent in-edges in the cyclic order of in-edges around v. (They may have out-edges between them.) Suppose e^*, the dual edge crossing e, has a lower postorder number as found by Algorithm 3 than f^*, the dual edge crossing f. Then the ET depth first search of G starting from the source with the designated first edge will visit edge e before edge f.*

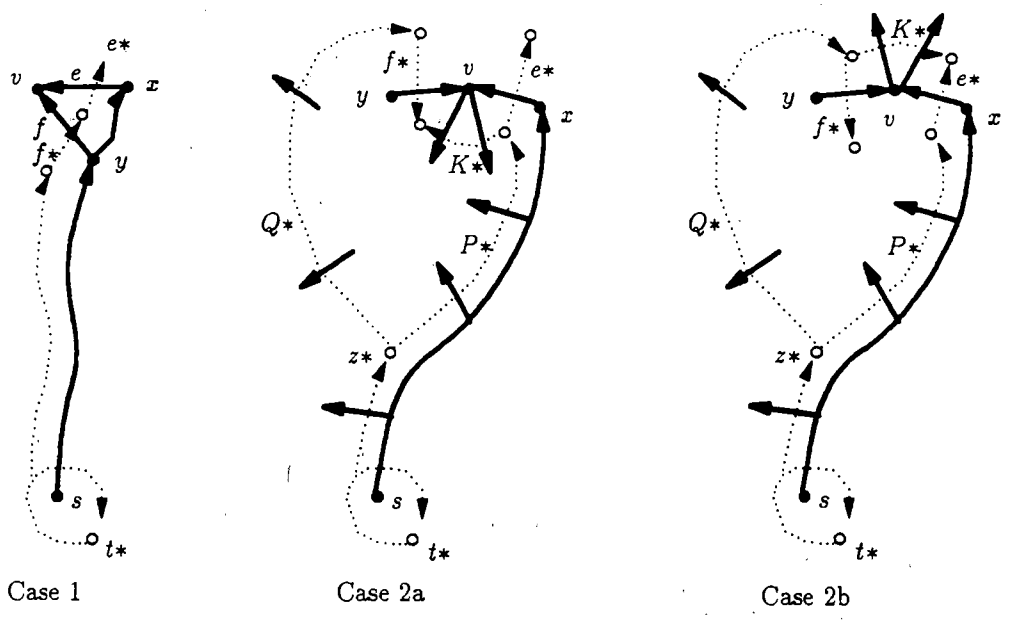


Figure 12
Illustration for Theorem 3.6

Proof. (Figure 12) There are two possible cases, depending on the relationship between e^* and f^* on the RHOP tree.

1. Both dual edges are on the same RHOP path from the root.

In this case, f^* must be the edge closer to the root in order for the postorder number of e^* to be less that that of f^*. Let e be the edge from vertex x to v, and f be the edge from y to v. By corollary 3.4 there is a path P in G from the source to y and then to x lying to the right of the RHOP path. If x is visited before y by the ET dfs tour of G, edge e will be visited before edge f. Consider the case where y is visited by the ET dfs tour before x. Again because the ET dfs tour visits all out-edges in a counterclockwise order, the edge along the path from y to x and all vertices and edges reachable from it, including edge e, will be visited before edge f.

2. The dual edges are on different branches of the RHOP tree.

Again, let e be the edge from x to v, and f be the edge from y to v in G. If there are out-edges from v between edges e and f, let K^* be the RHOP path segment which crosses them all. If the clockwise cyclic order of edges incident on v has in-edge e followed immediately by the out-edges crossed by K^* followed immediately by in-edge f, then K^* will start at the tail of e^* and end at the head of f^*. We will call this case 2a. If we switch the ordering of e and f, K^* will point from the tail of f to the head of e. This will be case 2b. If there exist no out-edges between e and f, we have the degenerate cases of 2a and 2b. In the degenerate case 2a, having no out-edges between e and f means that the tail of e^* is the head of f^*. In the degenerate case 2b, the tail of f^* is the head of e^*.

Let P^* and Q^* be the RHOP paths to the tails of e^* and f^* respectively from their lowest common ancestor z^* on the RHOP tree. Then in case 2a, P^*, Q^*, K^* if it exists, and f^* together form a closed boundary B. In case 2b, P^*, Q^*, K^* if it exists, and e^* form a closed boundary B. If we define the region inside the root cycle as the outside face, so that it is outside the closed boundary B, then both the source and the RHOP tree path from the root cycle to z^* are outside B. Since e^* has a lower postorder number than f^*, the path P^* to e^* will be to the right of the path Q^* to f^* at vertex z^*. This implies that the region to the right of P^* is outside B and the region to the right of Q^* and f^* is inside B. Since by definition of the dual edges, the tail of any edge g is to the right of its associated dual edge g^*, x is outside and y is inside the closed boundary B.

Because of the definition of an RHOP path, all paths in G from the source to y must cross the boundary formed by the RHOP tree path P^* to e^*. (They cannot cross K^* since that would create a cycle in G.) Since by corollary 3.4, the RHOP path to e^* is paralleled on the right by a path P in G from the source to x, the dfs tree path from the source to y in G must include on it some vertex u on P. We can now argue as in part 1 that edge e must be visited by the ET dfs tour before the edge out of u which crosses the RHOP boundary B. Thus e is visited before edge f. ∎

Theorem 3.6 thus shows that Algorithm 3 chooses the correct in-edge for an ET dfs tree in step 6. Each step in the algorithm can be done in $O(\log n)$ time using $O(n)$ processors in a EREW PRAM model, and thus Algorithm 3 can be done with the same time and processor bounds.

CHAPTER 4
ET dfs for planar dags with multiple sources and sinks

The ET depth first search of a planar dag with multiple sources and multiple sinks can be done if the dag can be partitioned into an ordered set of subgraphs, each subgraph containing a single source and vertices which are reachable from that source and possibly also from sources contained in subgraphs after it in order. Given such a partition, we can run the algorithm in the previous chapter for planar dags with a single source on each subgraph separately to find the depth first forest of the original graph. This chapter gives an algorithm for such a partitioning which runs in $O(\log k \log n)$ time for a graph with k sources in a CRCW model using a linear number of processors. We also give a simpler algorithm which will find all vertices reachable by a single source. If we are given a specific source, which we will call the root source, it will be the source for the first partition set and hence the root for the lowest numbered tree in the forest.

4.1. Informal description of the partitioning algorithm

The partitioning algorithm divides the vertices of a graph G into ordered family of partition sets, one partition set for each source in G. For every vertex v in a given partition set, there exists a path to v from the source for that partition set. The order among the partition sets is such that if vertex v is in partition set i, there can exist paths to v from sources for partition sets j, where $j > i$, but there exist no paths to v from sources for partition sets k, where $k < i$.

The algorithm works in a manner very similar to Knuth's topological sort [Knu73], with the modification that instead of removing a vertex at a time, we remove sets of vertices. Suppose in a given step, we remove in parallel all sources (vertices with no in-edge) present at that time with the exception of the root source. For each non-root source we also remove in the same step, all vertices which can be reached only from that source and from no other source. (We will refer to these as *attendant vertices* for that source.) Two different situations can then occur. In one case, we can remove a source and its attendant vertices without helping to create any new sources: all vertices to which they were adjacent still have in-edges in the modified graph. In such a case, the source will be considered inactive; no new vertices will be added to its partition set.

In the second case, the removal of a source and its attendant vertices contributes to the creation of a new source. First we note that the removal of more than one source and its attendant vertices must have contributed to the creation of a new source x. Suppose that x originally had in-edges from old sources in a set $\{y_i\}$ or from their attendant vertices. We pick one old source y_j out of the set by some priority rule and assign the new source x to y_j. The new source x will inherit the identity of y_j, including its priority ordering. We will call such new sources *fake* sources. If any old source has a new fake source assigned to it, we will call it *active*. We then replace the active old source y_j and

its attendant vertices by a new vertex y'_j, which also inherits the identity of the old source y_j, including its priority label. The new source y'_j will inherit all the out-edges of the old y_j and of its attendant vertices which point to vertices still present in the modified graph with the exception of the out-edges pointing to any new fake sources including x. We will call new sources such as y'_j new *simple* sources.

If any old source contributed to the creation of new fake sources, but was never picked, it will be considered inactive. No source in the new modified graph will have the identity of an old inactive source. The depth first ordering of the trees in the dfs forest can be obtained by taking the sources in reverse order from which they, and all new sources inheriting their identity, became inactive.

At the end of a given step as described above, there is no guarantee that the number of sources in the modified graph will have been reduced. In fact, the number of sources may have increased. We define two sources to be *contiguous* if they are on the boundary of one face. This means that in the dual graph, the positive faces surrounding the sources intersect at one dual vertex. A *cluster of contiguous sources* is defined as a set of sources whose positive faces form a connected component in the dual. We modify the algorithm further by considering each cluster of contiguous sources to be one new *supersource* for the succeeding step. (These are the *positive clusters* in [KS90].)

When an old source and its attendant vertices are contracted into one new simple source vertex y'_j in a given step, the new simple source will be contiguous with all new fake sources x to which the old source and its attendant vertices had out-edges. Similarly every new fake source x will be contiguous with all new simple sources y'_k corresponding to old sources y_k and their attendant vertices which had out-edges to x. Thus, one old source will be part of no more than one new supersource, but each new supersource must have had at least two old sources contributing to its existence. We note that if any new source inherits the identity of an old source, it will be part of a supersource. Thus if we count a supersource as one source, the number of sources at a given step must be at most half the number of sources at the preceding step, giving a bound of $O(\log k)$ for the number of steps.

The modification adds a complication. A given supersource s is actually a set of sources from the previous step, and the out-edges of the supersource s could be from different sources. The set of vertices which are reachable only from s must therefore be further partitioned into the partition sets for the sources making up s. We therefore need another partitioning procedure (procedure *AssignSupersourceVertices*) specifically for the vertices reachable from a supersource. If the boundary cycle for s becomes part of a new active source for the succeeding step, we can treat the new active source as a supersource s'. We will then need to use procedure *AssignSupersourceVertices* on the vertices reachable from s', using the information gained in the previous application of the procedure to assign each out-edge of s' to one of the original sources. Since the supersources are active at most $O(\log k)$ iterations, this process will be repeated at most $O(\log n)$ times in order to assign every vertex reachable from a supersource to one of the original sources of the graph.

4.2. Separating out a source and its attendant vertices

Instead of finding the boundary of the exact subgraph consisting of a given source and its attendant vertices, we find a clockwise dual cycle which serves the same purpose. The designated root source and the first edge out from it will determine the outside face: the outside face for G lies to the right of the first out-edge, and for the dual graph G^*, the outside face will be that associated with the designated root source in G.

Definition. Suppose we are given a graph G with multiple sources. Let one source be the designated root source, all other sources be non-root sources, and the outside faces for G and for G^* be defined as above. Then if we find a set of clockwise cycles in the dual graph G^*, one clockwise cycle C for each non-root source S, and the cycles have the following properties, then each C in the set is called a *boundary cycle* for its associated source S in G.

1. The associated source S is inside C.
2. Every vertex of G with an out-edge which crosses C can be reached from S by a path in G which lies entirely inside C.
3. No vertex which is reachable only from the sources inside C lies outside C (this implies all attendant vertices of S are inside C).
4. If S_1 and S_2 are two different sources in the same graph, the cycle C_1 for S_1 and the cycle C_2 for S_2 have no vertices (and hence no edges) in common. ∎

The boundary cycles for all non-root sources in a graph are found using the RHOP and RHIP structures of the dual G^*. We use them to identify simple clockwise cycles enclosing a given source. If there is an edge in the dual which belongs to both the RHOP and RHIP c-trees for the same source S, that edge can identify a cycle C formed by the RHOP path from the root cycle to the edge, the RHIP path from the edge to the root cycle, and the path on the root cycle connecting the two. Unfortunately, the cycle may not be a simple one: the RHOP and RHIP paths may intersect. In the following subsections, we show how to exclude non-simple cycles and counterclockwise cycles.

4.2.1. Classifying cycles formed by RHOP and RHIP paths

We will use the following properties of the RHOP and RHIP paths to classify the types of intersections of an RHOP path and an RHIP path.

Property 4.1 *An RHOP path cannot cross an RHIP path from the left. Equivalently, an RHIP path cannot cross an RHOP path from the right.*

This follows immediately from Property 2.1.

Property 4.2 *(Figure 13) Suppose edge e is on an RHOP c-tree T and on an RHIP c-tree U, where the two c-trees may or may not share the same root cycle. If we follow the RHIP path Q in U toward its root cycle, we will be following an RHOP path P in T until it ends in a leaf. The same is true if we follow the RHOP path P in T backwards toward the root cycle; we will be following an RHIP path Q in U backwards until it ends in a leaf. In other words, once an RHIP path shares an edge with an RHOP subtree, it will not leave that RHOP subtree until it follows some branch to its leaf; similarly if an RHOP path shares an edge in an RHIP subtree.* ∎

Again this is a consequence of Property 2.1.

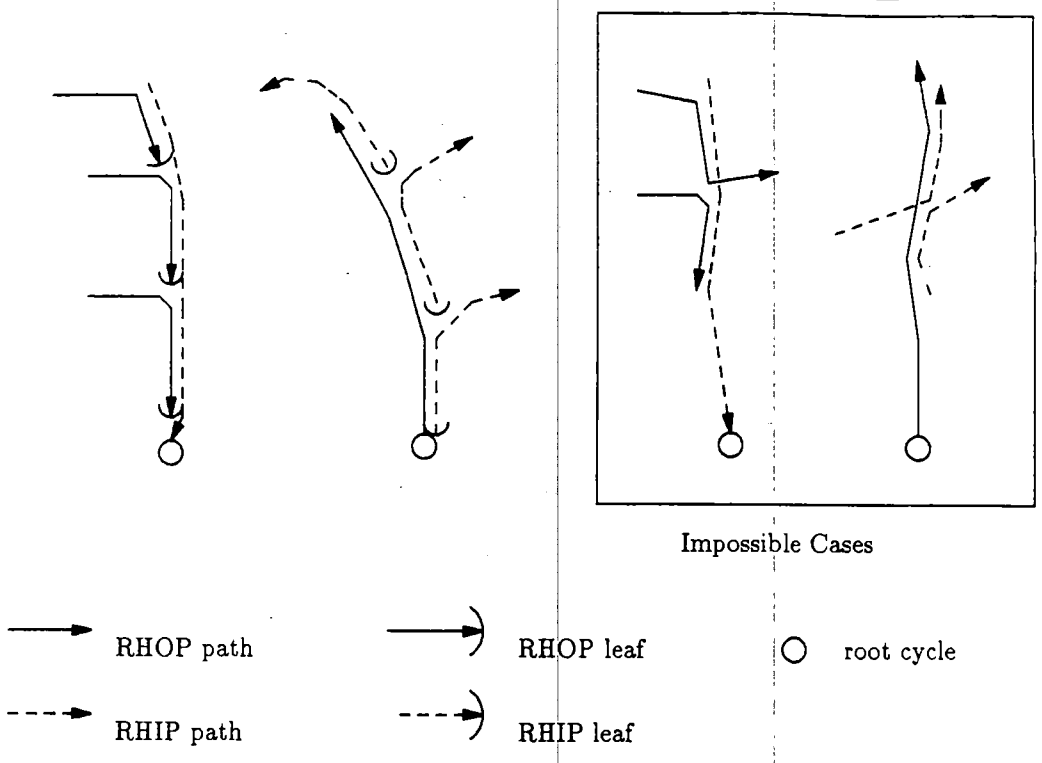


Figure 13
Illustration for Property 4.2

We now classify different types of intersections. We define an edge which is on both the RHOP c-tree and the RHIP c-tree of the same source S as an *overlap* edge for S.

Property 4.3 *(Figure 14) Let the RHOP path to an overlap edge e from the root cycle be P, and the RHIP path from e to the root cycle be Q. Let v be a vertex at which P and Q intersect. Note that by Property 4.1, P must cross Q from right to left. Then we can state the following:*

1. *If v is the intersection vertex closest to e on Q, it is the intersection closest to e on P and vice versa. (We are assuming that the paths only share vertices and not edges. If the paths share edges, then we can let v represent a path segment with little change in the arguments below.)*
2. *At v the in-edge on P and the out-edge on Q are both overlap edges. (If v represents a path segment as noted in item 1, this holds for all vertices on the segment, in particular the end points.)*
3. *If the intersection vertices (or intersection path segments) of P and Q are ordered $u_1, u_2, \ldots u_k$ on Q from the one closest to e to the one furthest from e and closest to the root cycle, they would be ordered in the same way on P.*
4. *Consider the simple cycle C created by the path from e to v on Q and then back to e on P, where v is the intersection vertex closest to e as in item 1. Then the root cycle must be to the right of the path of the cycle C. Depending on the relative*

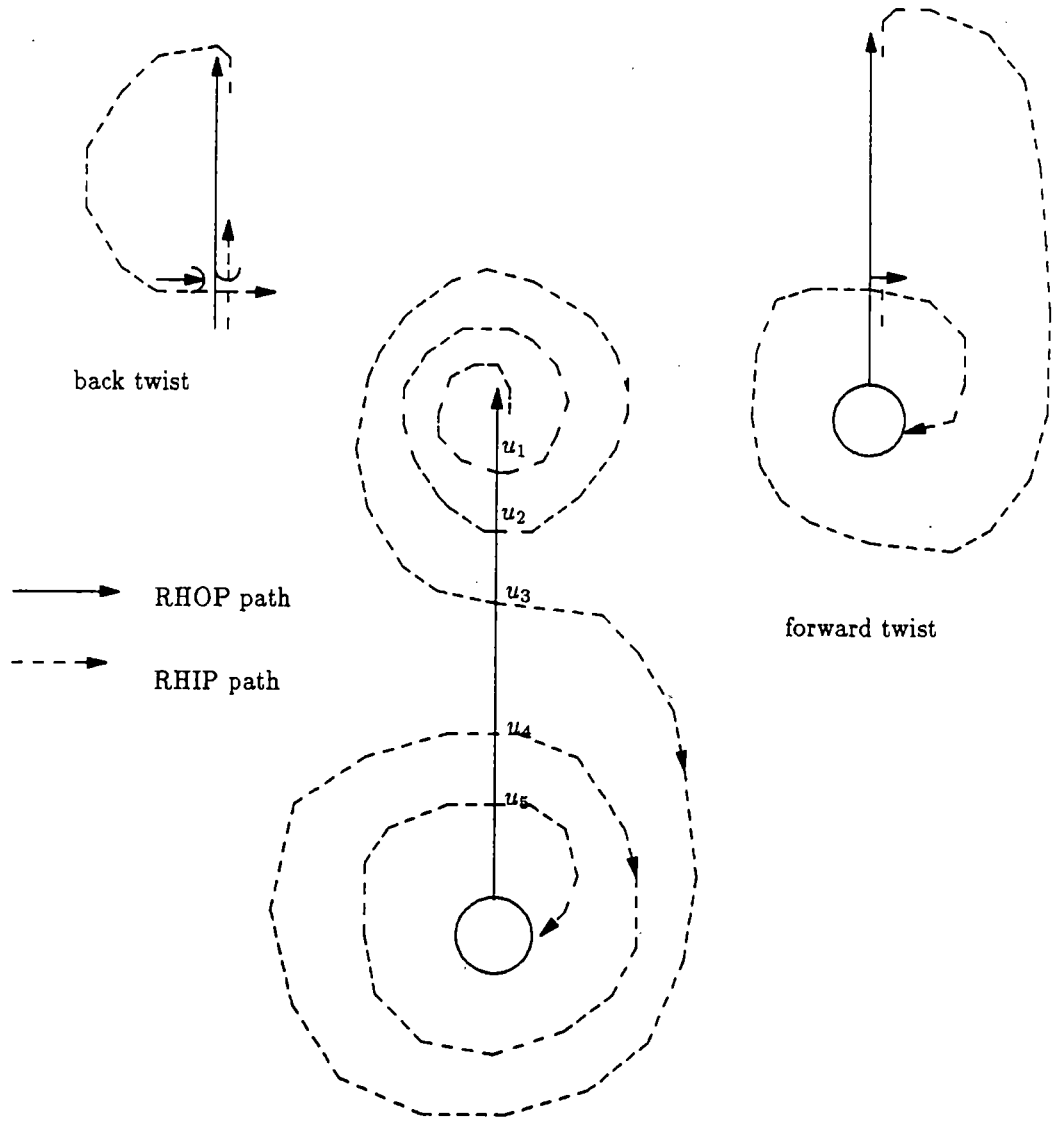


Figure 14
Classification of intersections of RHOP and RHIP paths

location of the outside face, C may be counterclockwise with the root cycle and source outside, or it may be clockwise with the root cycle inside. We will call the clockwise cycle a forward twist, and the counterclockwise cycle a back twist.

5. *On an ordered list of intersection vertices as in item 3, we can apply the classification of item 4 to the simple cycle formed by the path segments of P and Q which lie between the overlap edges incident at u_i and the next intersection vertex u_{i+1}. Note that if such a cycle defined by the pair (u_i, u_{i+1}) is a forward twist, all cycles defined by pairs closer to the root cycle must also be forward twists.* ∎

Proof Sketch.

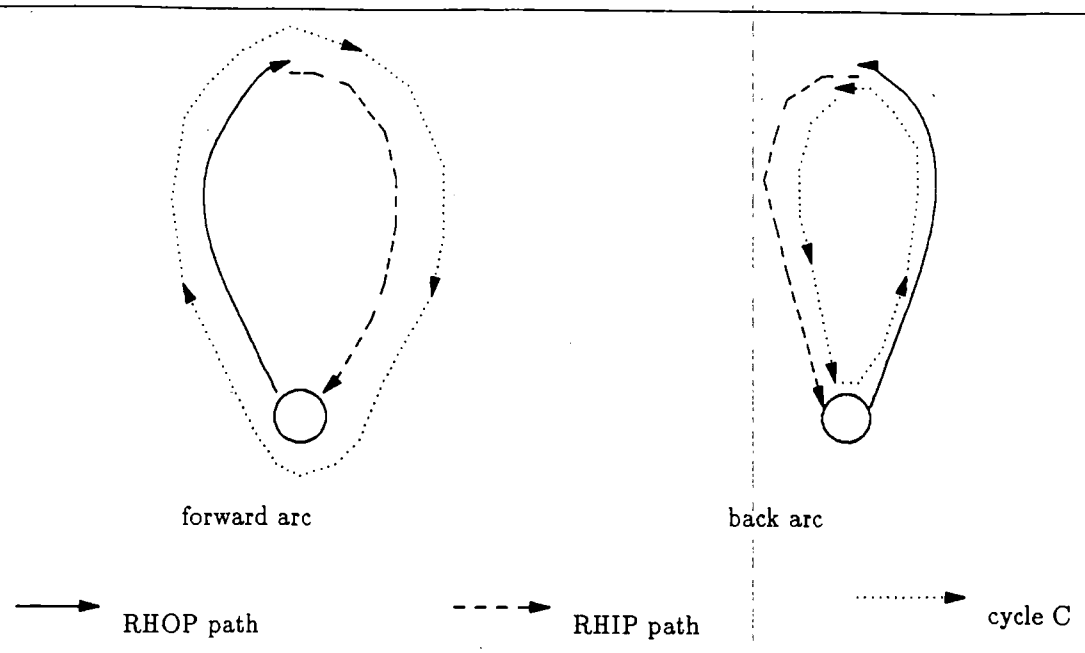


Figure 15
Forward and back arcs

1. This is due to Property 4.1: once we have a simple cycle from e to v on Q and from v back to E on P, the part of the path Q from v to the root cycle cannot cross back into the simple cycle. Similarly, the part of P from the root cycle to v cannot have previously crossed out from the simple cycle.
2. This is from the definitions of RHOP and RHIP.
3. This follows from the previous two items.
4. By Property 4.1, P must cross Q from the right to the left. Hence the root cycle must be to the right of the simple cycle C.
5. This is due again to Property 4.1 and to fact that two RHOP paths cannot cross; nor can two RHIP paths cross. ∎

We now consider the case when the paths P and Q do not intersect.

Property 4.4 *(Figure 15) Suppose the RHOP path P and the RHIP path Q that go to and from the overlap edge to the root cycle do not intersect. We will say that the pair of paths P and Q forms a forward arc if the simple cycle C formed by Q, the edges on the root cycle connecting the endpoints of Q and P, and P is clockwise. If C is counterclockwise, we will call the pair of paths a back arc. Note that since the root cycle is always clockwise, the simple cycle associated with a forward arc encloses the source associated with the root cycle. The cycle associated with the back arc does not.*

We finish the task of classifying the relationship of P and Q of overlap edges with the following:

Property 4.5 *If we have an ordered list of intersection points on P and Q as in items 3 and 5 in Property 4.3, and the pair closest to the root cycle defines a forward twist, then*

the arc defined by the intersection point closest to the root cycle must be forward. If the pair closest to the root cycle defines a backward twist, the arc may be forward or back.

Proof Sketch. Again by by Property 4.1, once a positive twist defines a clockwise cycle with the root cycle inside, the RHIP path cannot cross the cycle to get outside. A back twist presents no such obstacle. ∎

4.2.2. Classifying leaf orientations of an RHIP c-tree

In order to classify the different P and Q orientations defined by an overlap edge, we will find it useful to consider the relative orientation of a leaf of an RHIP c-tree, when the leaf has been cut off from a vertex on the same c-tree. Suppose that a leaf edge (u, v) and the vertex u are both on the same RHIP c-tree. We first note that the edge (u, v) must have been incident on the left side of the RHIP path that u is on by Property 2.1. The following classifies the relative orientations.

Property 4.6 *Let leaf edge $e = (u, v)$ and vertex u be on the same RHIP c-tree. Consider the edges on the c-tree to be undirected. Then there exists a path M from e to u consisting only of edges from the undirected c-tree. This path will be unique if we stipulate that the path from e to u will always go clockwise around the root cycle if the path touches the root cycle. Since $e = (u, v)$ and u are connected in the incidence graph of the dual by edge (u^i, e^i), we can define a directed cycle C using the undirected edges of the dual incidence graph going from e to u along the tree edge path M and then back to e along (u^i, e^i). We will define the relative orientation of e and u to be positive if C is a clockwise cycle and negative if C is a counterclockwise cycle. We will call C the leaf orientation cycle or lo cycle for the edge-vertex pair.* ∎

In order to make certain that our procedure for finding leaf orientations is correct for all cases, we will enumerate all possible ways that e and u can be related on the RHIP c-tree.

Property 4.7 *(Figure 16) Let Q be the RHIP path from edge $e = (u, v)$ to the root cycle and R be the RHIP path from u to the root cycle. If Q and R intersect, let x be the lowest common ancestor of e and u in a c-subtree of the RHIP c-tree. There are three cases, one with subcases.*

1. *Q and R intersect, and x is distinct from u. The lo cycle C will in this case go forward along Q from e to x and backward along R from x to u.*
   a. *If the root cycle is outside C, we will call this case 1A. Depending on whether C is clockwise or counterclockwise, we will call it positive 1A or negative 1A. Since the root cycle is outside C, the RHIP path from x to the root cycle must lie to the left of C if C is clockwise and to the right of C if C is counter clockwise. This means that at x, the in-edge on R is to the right of the in-edge on Q in the relative orientation of in-edges for the positive 1A case. For the negative 1A case, the in-edge on R is to the left of the in-edge on Q.*
   b. *If the root cycle is inside C, it will be called case 1B. Again, we will have positive 1B and negative 1B depending on whether C is clockwise or counterclockwise. Since the root cycle is now inside, the relative orientation of R and Q at x will be reversed for the two cases: for positive 1B, R is to the left of Q and for negative 1B, R is to the right of Q.*

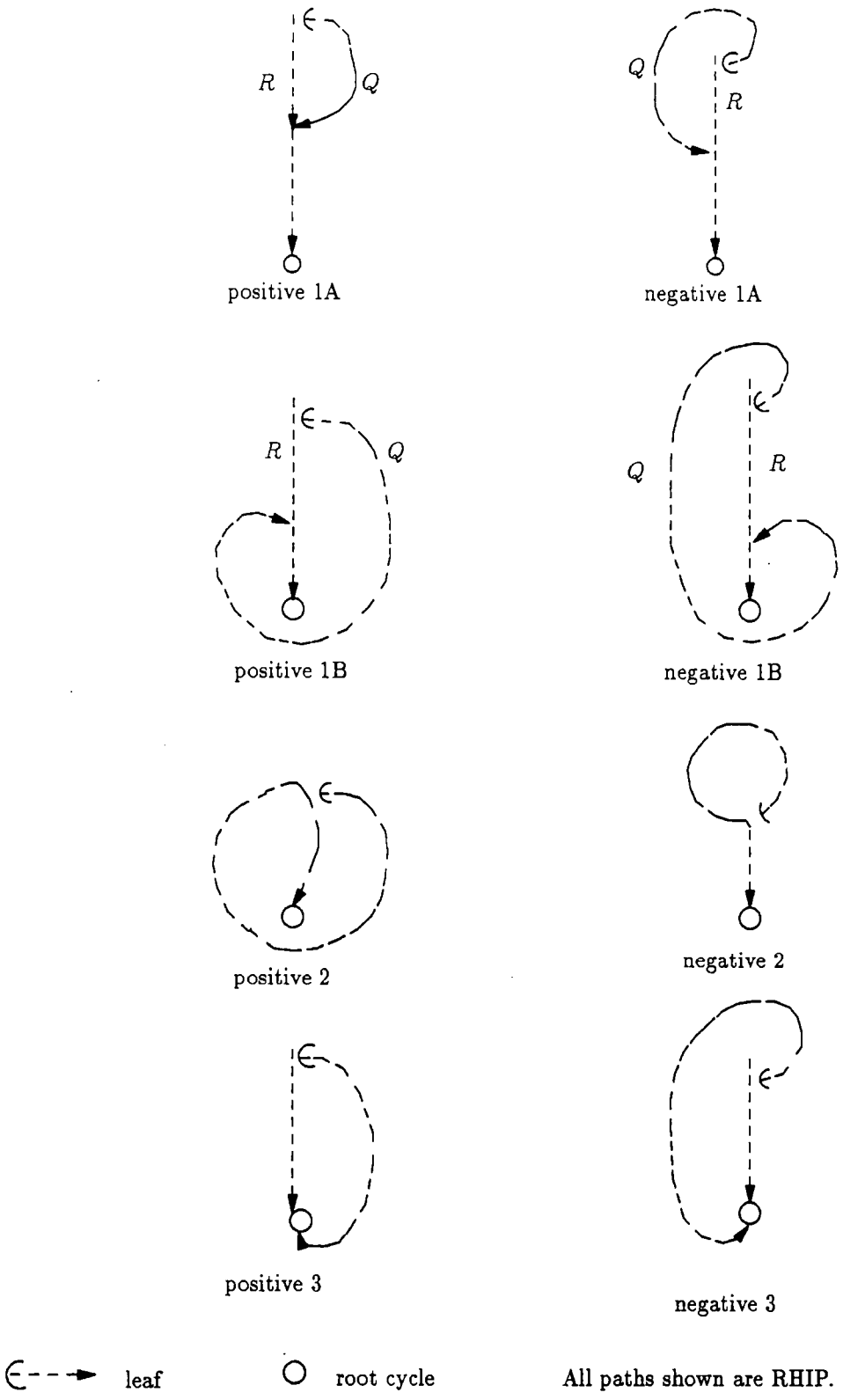


Figure 16
Classification of RHIP leaf orientations

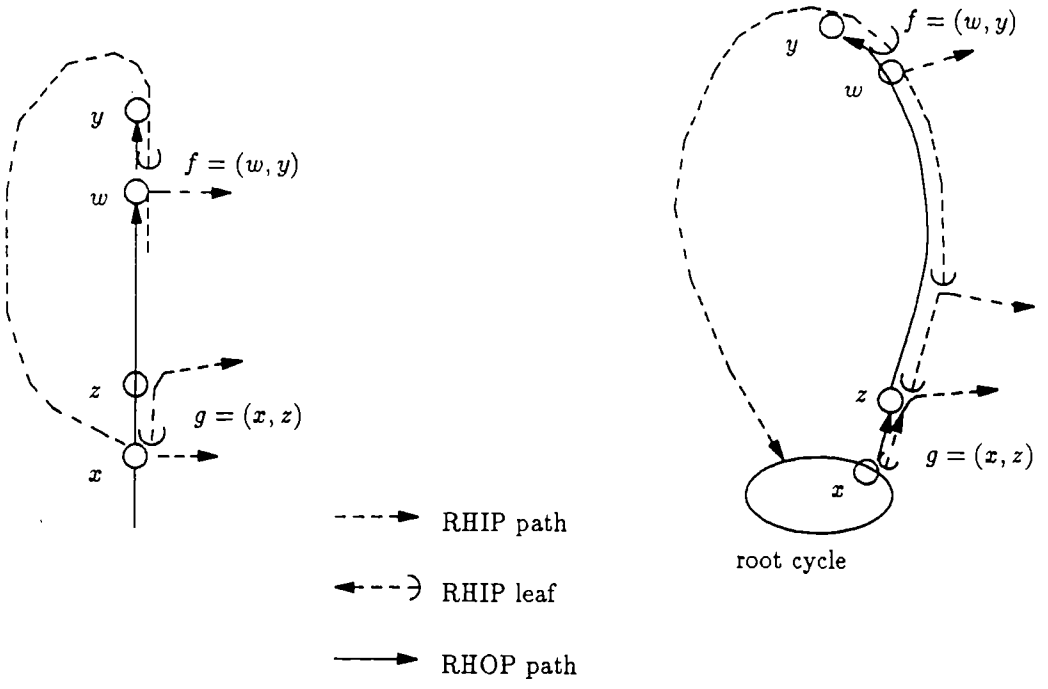


Figure 17
Illustration for Theorem 4.1

2. *Q and R intersect in such a way that x, the common ancestor of e and u, is the vertex u. Then the lo cycle C will be the cycle formed by the part of the RHIP path Q from e to u. Since edge e must come from the left side of Q by Property 2.1, the path to the root cycle from u must always lie to the right of C. Again C can be clockwise or counterclockwise, giving cases positive 2 and negative 2.*

3. *Q and R do not intersect. The lo cycle C is then formed by following Q forward to the root cycle, following the root cycle edges to the vertex at the end of R, and following R backward to u. We will include in this the case that u is on the root cycle; this is just the degenerate case when R has been reduced to a vertex. We note that the source associated with the root cycle will always lie to the right of C. Again, we have positive 3 and negative 3, depending on whether C is clockwise or counterclockwise.*

4.2.3. Separating forward twists and arcs from back twists and arcs

The following theorem shows how finding the relative orientations for RHIP leaves will help distinguish forward twists and arcs from back twists and arcs. **Theorem 4.1** *Suppose we are given the counterclockwise cycle associated with a back twist or arc. It will be composed of a section P of an RHOP path, a section Q of an RHIP path, and possibly part of the root cycle. Assume that the entire RHOP section P is composed of overlap edges. Then among these edges will be at least one edge, which in the RHIP c-tree will be a leaf oriented in one of the negative cases (negative 1A, 1B, 2, or 3).*

Proof. (Figure 17) Consider the case of P and Q intersecting in a back twist. If the intersection is only at one vertex, let x be that vertex at which P intersects Q. If P and Q intersect in a path segment, let x be the end of the segment closest to the overlap edge, i.e., the vertex at which P and Q diverge again after intersecting. Now consider the case where P and Q form a back arc. In this case, let x be P's end vertex on the root cycle.

Suppose we start at the overlap edge and go backward on P toward x. By Property 4.2, we will be following an RHIP path backward until we come to a leaf. Let the leaf be edge $f = (w, y)$. Then w is the vertex on P which is on some new RHIP path. Since we are assuming that all edges on P are overlap edges, this new RHIP path belongs to the same RHIP c-tree as the RHIP leaf f. Thus we can classify the orientation of f according to Property 4.7. We can again follow the new RHIP path backward along P, and repeat for the next RHIP leaf that we come to. This is continued until we reach x on P. Since the edge $g = (x, z)$ (the edge on P out of x) will be a leaf edge for the RHIP c-tree, we are guaranteed that there will be at least one such leaf between the overlap edge and x on P.

We now show that at least one of the RHIP leaves found above will have a negative orientation. If there is only one leaf edge along P at x, the same RHIP path goes from $g = (x, z)$ to the overlap edge and back to x, forming a cycle. Since this is a back twist or arc, it is the negative 2 case or the degenerate negative 3 case. If there is more than one RHIP leaf edge along P, we give a proof by construction. Suppose for each leaf-vertex pair along P, we define a cycle C to be that formed by following the RHIP path Q from the leaf to where it first joins the RHIP path R from the vertex. (This is the lo cycle C defined in Property 4.6.) For the purpose of this argument, we can consider the root cycle to be a single vertex, so that if we have case 3 of RHIP leaf orientation, Q and R will be assumed to meet.

If we look at the edges of the RHIP c-tree which lie on the C cycles, we get a subgraph of the c-tree which itself forms a tree or a c-tree, where the root, if it is not the root cycle, is the lowest common ancestor of all the leaves. Topologically speaking, this is equivalent to having a counterclockwise cycle, a point p outside the cycle, and paths which go from the the cycle to p. (Figure 18) We note that the regions lying to the right of the C cycles are disjoint, since the RHIP paths cannot cross each other, and that together they cover the entire plane (or surface of the sphere) except for the region inside the counterclockwise cycle defined by the back twist or arc (and possibly the inside of the collapsed root cycle). Since the back twist cannot contain the outside face inside, by definition of back twist, the outside face must be in a region lying to the right of one of the C cycles. By definition of counterclockwise, that C cycle must be counterclockwise, giving rise to a negative leaf orientation. ∎

The following lemma is then an immediate consequence of Theorem 4.1.

Lemma 4.2 *Suppose we are given a path P on an RHOP c-tree with its starting endpoint on the root cycle. Let all edges on P be overlap edges. Then if none of the edges on P have a negative RHIP leaf orientation, then all overlap edges on P define either positive arcs or positive twists.*

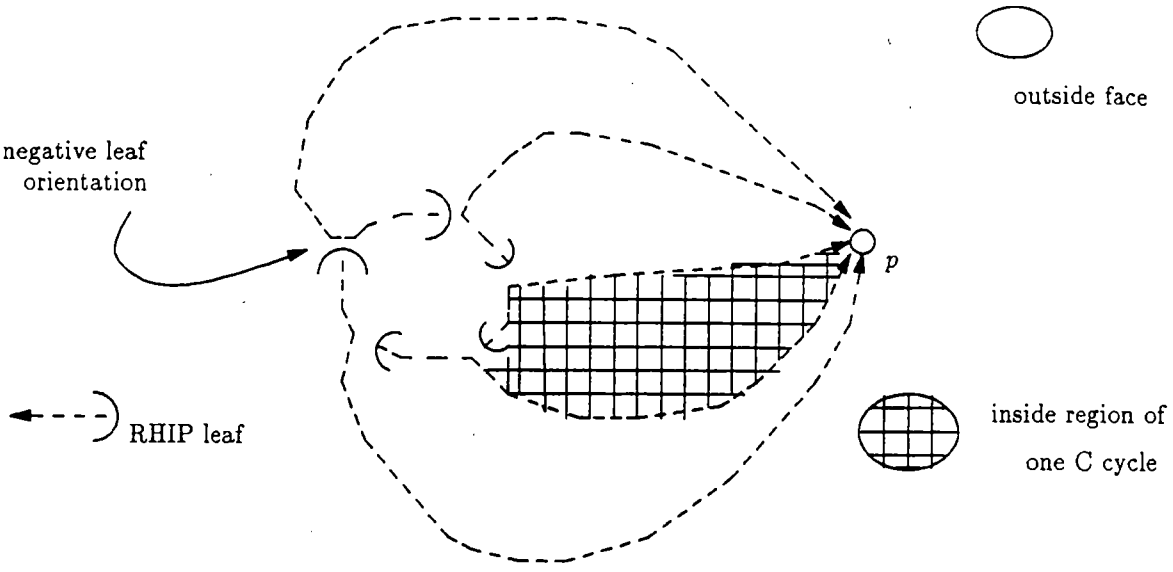


Figure 18
Orientation of the RHIP paths in proof of Theorem 4.1

4.2.4. Finding the leaf and cycle orientations

In this section we present the procedure for classifying the orientation of a given leaf edge $e = (u, v)$ in an RHIP c-tree according to Property 4.7. For this and for the procedure for finding cycle orientation given in this section, we need to check if a given node u is an ancestor of another node v in a tree. We can easily do this in constant time if the preorder and postorder numbers of the the nodes in the tree are available by using the following well known lemma [BB88, p. 208].

Lemma 4.3 *Given two nodes u and v in a tree, u is an proper ancestor of v if and only if both conditions below are met.*

$$preorder(u) < preorder(v)$$

$$postorder(u) > postorder(v)$$

∎

Determining for a pair of paths in a c-tree if one path is incident on another from the left or the right is also easy if we have assigned the preorder numbers using a right-hand Euler tour of the c-tree.

Lemma 4.4 *Let Q be the path from a vertex u to the root cycle in an RHIP c-tree. Let R be the path from a vertex v to the root cycle, where u and v are in the same c-subtree of the RHIP c-tree. Let x be the lowest common ancestor of u and v in the RHIP c-subtree. At x, let the in-edge on R be to the right of the in-edge on Q. Then a right-hand Euler tour of the RHIP c-subtree will visit u before v.* ∎

The proof is a straightforward application of the definition of a right-hand Euler tour, remembering that since the c-subtree is an in-tree, all tree edges and paths are pointing toward the root.

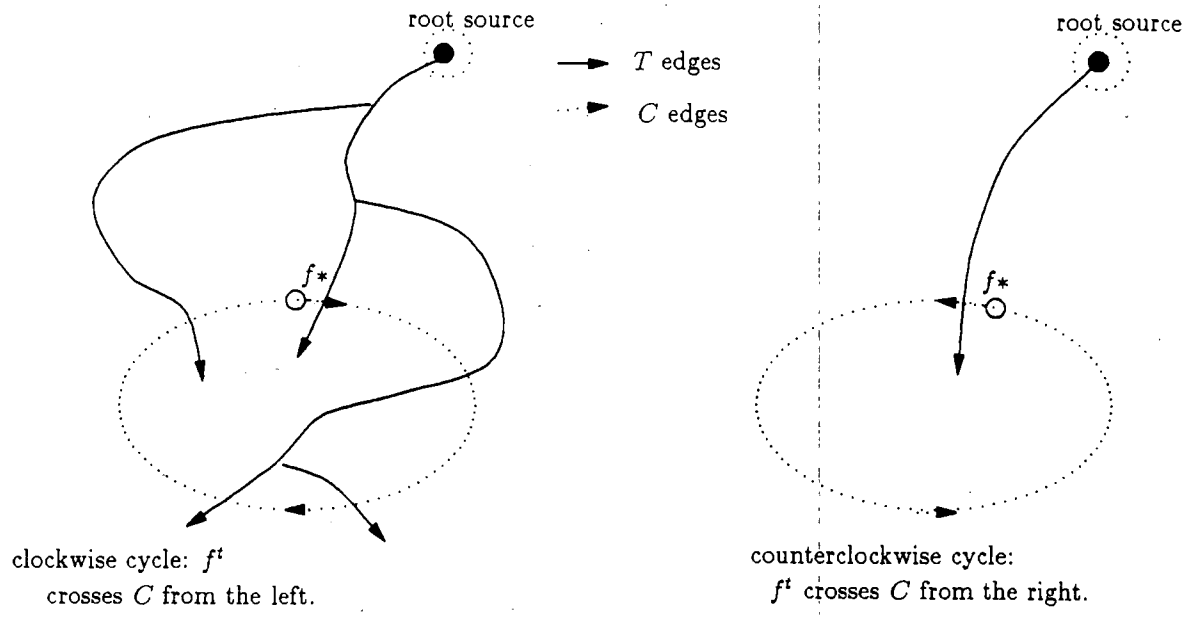


Figure 19
Illustration for Procedure *CycleOrientation*

Given these two lemmas, separating out the cases 1A, 1B, 2, and 3 is straightforward. To separate the cases into positive and negative, we need to find the orientations of the lo cycle C as defined in Property 4.7. We will first give a general procedure for finding if a cycle is clockwise or counterclockwise. The procedure for classifying the leaf orientation will use an adaptation of the general procedure.

The idea behind the procedure to find the cycle orientation is to find some path M starting from the outside face and crossing the cycle for the first time at an edge f^* on the cycle. By checking if M crosses the cycle from the left or from the right, we can tell whether the outside face is to the left or the right of the cycle, and hence if the cycle is clockwise or counterclockwise. We find such a path M by looking at the tree paths for a spanning tree of the *undirected* graph, with the root source enclosed by the outside face as the root of the spanning tree. We can ensure that the tree path M has not previously crossed the cycle by finding the edge crossing the cycle which is closest to the root source.

procedure *CycleOrientation*
(*Given a simple cycle C in a dual graph G^*, this procedure will determine if the cycle is clockwise or counterclockwise. The outside face in G^* will be the one corresponding to the root source in G. *) (Figure 19)

1. Consider the undirected version G^u of the original graph G. Find a spanning tree T of G^u and orient it as an out-tree with the root source of G as the root of T. Note that the direction of an edge e^t in T has no relationship to the direction of the same edge e as considered as an edge of G.
2. Find for every edge in T its level order, i.e. its distance from the root of T.

3. For every dual edge on C, check to see if it is crossed by an edge in T. (More formally, we see if it is a dual edge e^* in G^* corresponding to an edge e in G such that its undirected version e^u was chosen to be in T and oriented to be e^t.) If so, find the level order of the edge in T. There must exist at least one such edge since there must be at least one vertex of G inside C.
4. By pointer doubling, find the edge f^* on C crossed by an edge f^t in T with the lowest level order of all tree edges crossing C. If there exists more than one such edge, pick the one by some rule such as the lowest edge number. We now know there exists some path on T from the root source to the cycle C crossing at f^* which has not previously crossed C.
5. Find the relative orientation of the f and f^t. If they are going in the same direction, f^t is crossing the edge f^* of C from the right to the left. Then the outside face as determined by the root source will be to the right of C, and the cycle C is counterclockwise. If f and f^t are going in opposite directions, the cycle C is clockwise.

For step 1, there exists an optimal CRCW algorithm using $O(n/\log n)$ processors and $O(n)$ time for finding and orienting a spanning tree for a planar undirected graph [Hag88]. All other steps can be done in $O(\log n)$ time with $O(n)$ processors in the EREW model. The same bounds will hold for finding many cycle orientations in parallel as long as the cycles are disjoint.

We cannot use this procedure without modification to find many leaf orientations in parallel, since the lo cycles defined by leaf-vertex pairs will, in general, share edges. We will instead, preprocess a c-tree as a whole, finding for every c-tree path from a c-tree edge to the root cycle, the edge first crossed by a path M starting from the root source. We again use a spanning tree of the undirected graph to find such a path M. By looking at the possible cases of how such a M can cross the RHIP c-tree paths, we can classify the leaf orientation.

procedure *CheckLeafOrientation*

(*Given an RHIP c-tree, find the leaf orientation for all leaf-vertex pairs, where leaf $e = (u, v)$ and vertex u are both in the RHIP c-tree. Let Q be the RHIP path from e to the root cycle and R be the RHIP path from u to the root cycle.*)

1. Find and preprocess T as in steps 1 and 2 of Procedure CycleOrientation.
2. For every edge on the RHIP c-tree, check to see if it is crossed by an edge in T. If it is, record the level order of the crossing T edge.
3. Find for every vertex w or edge g on the c-tree, the edge on the RHIP path from w or g to the root cycle that is crossed by the lowest level order T edge. If no edge on the RHIP path is crossed by a T edge, indicate this for w or g. The level order of such a non-existent T edge can be defined to be $n + 1$. Also find the T edge $root^t$ crossing into the root cycle.
4. Using the ET technique, find for all vertices and edges in the RHIP c-tree, the preorder and postorder numbering. For this step, the root cycle may be regarded as one root vertex.

5. By pointer doubling, find for every vertex w and edge g in the RHIP c-subtrees, the vertex *c-root*(w) or *c-root*(g) on the root cycle which is the root of the c-subtree that w or g is in.
6. For every leaf $e = (u, v)$ and vertex u in the RHIP c-tree, do the following.
   a. Find f^t, the tree edge with lowest level order associated with e, and hence with path Q, using the precomputed data from step 3. Similarly, find h^t, the tree edge with lowest level order associated with u, and hence with path R.
   b. **If** (*c-root*$(e) \neq$*c-root*(u)) and (*levelorder*$(root^t)$ is less than either *levelorder*(f^t) or *levelorder*(h^t))

   (* Must be case 3 since e and u are in different c-subtrees. The lowest level order T edge crosses into the root cycle. When we have case 3, but the lowest level order T edge crosses either Q or R, it is handled in case (ii) and (iii) below.*)

   then if $root^t$ crosses the root cycle somewhere on the directed path from *c-root*(e) to *c-root*(u)
   then the orientation is positive
   else the orientation is negative.

   else
   Case *i.* $f^t = h^t$

   (* The lowest level order tree edge crosses the RHIP path to the root cycle from the lowest common ancestor of e and u; this implies case 1A or negative 2.*)

   if *preorder*$(u) <$ *preorder*(e)
   then the orientation is negative
   else the orientation is positive

   Case *ii.* *levelorder*$(f^t) \leq$ *levelorder*(h^t)

   (* The lowest level order tree edge crosses Q. This and the next case are the standard cases in which the lowest level order T edge crosses into the lo cycle C defined for leaf-vertex pairs by Property 4.6. This can happen in all cases 1, 2, and 3.*)

   if f^t crosses Q from the right
   then the orientation is negative
   else the orientation is positive

   Case *iii.* *levelorder*$(f^t) >$ *levelorder*(h^t)

   (* The lowest level order tree edge crosses R. Again, this is the standard case which can occur in all cases 1, 2, and 3. *)

   if h^t crosses R from the left
   then the orientation is negative
   else the orientation is positive

We note that since the lo cycle C must contain at least one vertex of G inside, at least one of f^t or h^t must exist for cases 1A, 1B, and 2. For case 3, if neither f^t nor h^t exists, then we know that $root^t$ must always exist and have level order less than $n + 1$.

The preprocessing steps 1 through 5 in Procedure *CheckLeafOrientation* can all be done using $O(n)$ processors and $O(\log n)$ time on a CRCW PRAM. For a given leaf-vertex pair, step 6 can be done in constant time by a single processor, so that the procedure can run in the same time bounds as procedure *CycleOrientation* for $O(n)$ such pairs on the RHIP c-tree.

4.2.5. Procedure for finding boundary cycles

The algorithm for finding the boundary cycle follows. It proceeds by finding forward arcs and twists for each root cycle, where all edges on the arcs and twists are overlap edges. Since these define clockwise cycles, by Property 2.2, we can use the LHOP structure to find the cycles bounding the regions inside the clockwise cycles. The outermost clockwise cycles out of these will be the boundary cycles. Each step takes at most $O(\log n)$ time using $O(n)$ processors on a CRCW PRAM, so we get the same bounds for the entire algorithm. We assume that all vertices in G^* are simple.

procedure *FindBoundaryCycle*

1. Find the RHIP and RHOP structures for the dual graph G^* and identify for all edges which c-trees they belong to for each structure.
2. Mark all edges as *overlap* if they belong to the RHOP c-tree and the RHIP c-tree for the same source.
3. Exclusive trim the RHOP and RHIP c-trees so that they consist only of overlap edges. This can be done using the Euler tour technique as explained in Chapter 2.
4. For every leaf of the trimmed RHIP c-tree which is also an edge in the trimmed RHOP c-tree, find the orientation of the RHIP leaf using Procedure *CheckLeafOrientation*.
5. If the orientation of the leaf is negative, mark the edge in the RHOP c-tree as *bad*. If the orientation of the leaf is positive, mark the edge in the RHOP c-tree as *good*. (Note that there may be unmarked edges on the trimmed RHOP c-trees. They may not be RHIP leaves at all, or they may be RHIP leaves but not on the trimmed RHIP c-tree. This can occur if some edge along the RHIP c-tree path from the RHIP leaf to the root cycle was not an overlap edge.)
6. Exclusive trim the RHOP c-trees again so that they contain no bad edges, i.e., they contain only non-bad edges. (By Lemma 4.2, the trimmed RHOP c-trees contain no overlap edges defining back arcs or back twists.)
7. Inclusive trim the RHOP c-trees for the last time, this time discarding any branch ends which contain no good edges. Again this can be done using the Euler technique. (This means that all edges on the trimmed RHOP c-trees are along some forward arc or forward twist, where all edges on such arcs and twists are overlap.)
8. Mark all good leaves in the trimmed RHIP c-trees which are also edges on the final RHOP c-trees. Inclusive trim the RHIP c-trees by discarding any edge which does not lie on a path between a marked leaf and the root cycle. (This ensures that all edges on the trimmed RHIP c-trees are also along some forward arc or twist, where all edges on such arcs and twists are overlap.)

9. Take the subgraph induced by the edges of the final RHOP c-trees and the RHIP c-trees and use the LHOP structure to find the outer cycles. (By Property 2.2, the cycles found bound the regions inside the forward arcs and twists.)
10. Use Procedure *CycleOrientation* to find the clockwise outer cycle for each RHIP and RHOP c-tree pair with the same source out of the cycles found in the previous step. (* We show in the theorem below that such a clockwise cycle exists. *) Mark it as the boundary cycle for the source.

We use the following theorem to show that the procedure finds boundary cycles for the sources.

Theorem 4.5 *A cycle C in G^* is found for each RHIP and RHOP c-tree pair by the procedure FindBoundaryCycle and satisfies the definition of a boundary cycle for the source S in G associated with the root cycle shared by the c-tree pair.*

Proof.

1. We first show that at the end of step 10 a clockwise cycle C will always be found for every source S, and that source S will be inside C. We thus show that the procedure will produce a cycle which has property 1 of a boundary cycle: it is clockwise and it contains S inside.

Consider the edges marked at the end of step 8. Every edge is an overlap edge by step 3. It is also part of a trimmed RHIP c-tree or part of a trimmed RHOP c-tree. By the inclusive trimming of the RHOP c-trees in step 7 and of the RHIP c-trees in step 8, all leaf edges of both trimmed c-trees are *good*. Thus every edge on the trimmed c-trees is on a forward arc or twist defined by paths on the trimmed c-trees. By definition of forward arcs and twists, the edges composing a forward arc or twist defines a clockwise cycle.

Consider the region R composed of the union of all regions inside the clockwise cycles defined by the edges of a particular pair of trimmed RHOP and RHIP c-trees sharing the same root cycle. Note that R is never empty, since the root cycle edges will always be part of the trimmed RHOP and RHIP c-trees at the end of step 8. (Inclusive and exclusive trimming never trim the root cycle edges.) Since all the clockwise cycles defined by forward twists and arcs contain the region inside the root cycle, R is a connected region. (We will later see that R will generally be a simply connected or hole-free region, but at this time we will see that the proof will follow even if there exist holes in R.) When we find cycles on the marked edges using LHOP in step 9, we are finding the boundaries of R with R lying to the right of the cycles. Since by definition, none of the clockwise cycles contains the outside face inside it, R does not contain the outside face. Then one and only one of the cycles which form the boundary for R found in step 9 must contain the outside face in the region to the left of the cycle path. This will then be the clockwise cycle C of step 10. Thus C exists, is clockwise, and contains the source S inside since R is inside C and contains the region inside the root cycle.

2. We will show that C satisfies Property 2 of a boundary cycle: there exists a directed path lying entirely inside C from the source S to any vertex with an out-edge crossing C.

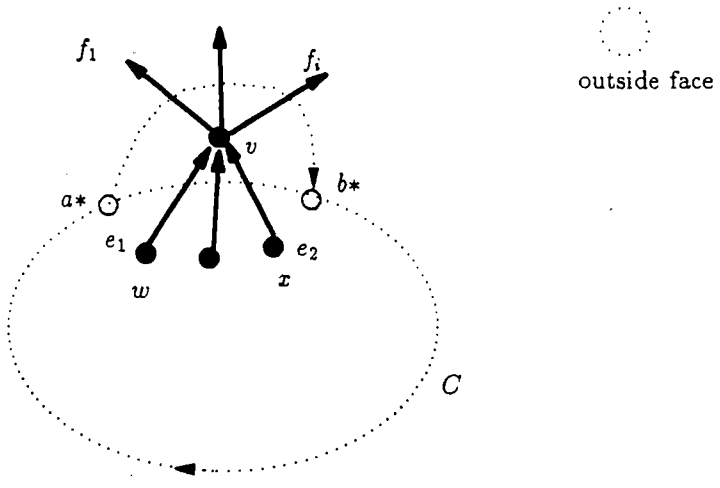


Figure 20
Proof of property 3 of the boundary cycle

As noted in 1 above, the cycle C encloses the regions inside the clockwise cycles defined by forward arcs or twists. Every edge on C is an overlap edge and by the corollaries 3.4 and 3.5, there must exist a directed path which lies completely inside the clockwise cycle defined by the associated forward arc or twist for the overlap edge and which starts at the source and crosses the overlap edge on C. Thus there is a directed path in G from the source to any edge crossing C, which lies entirely inside C.

3. We will now show that C satisfies Property 3 of a boundary cycle: no vertex which is reachable only from sources inside C lies outside C. (Figure 20)

This will be a proof by contradiction. We assume that there exists a vertex outside the cycle C which is reachable only from sources inside C. In particular, this means that there must be at least one vertex all of whose in-edges must be edges crossing C. Call this vertex v. Consider the outside face that would be formed if we were to remove all vertices except v and the vertices inside C. Let e_1 and e_2 be the two in-edges to v crossing C from vertices inside C which would be on the boundary of such an outside face. (What follows can also apply to the case when there is only one in-edge to v crossing C with minor modifications.)

We first assume that v has no out-edges between e_1 and e_2 in its clockwise ordering of edges. Let $e_1 = (w, v)$ where w is inside C, and $e_2 = (x, v)$ where x is also inside C. If v has no out-edge between e_1 and e_2, then w, v and x are all on the same face F in G. The vertex in the dual G^* which corresponds to F will then have a dual out-edge e_1^* crossing e_1 and a dual in-edge e_2^* crossing e_2. By assumption both e_1 and e_2 cross C so that both e_1^* and e_2^* are on the simple clockwise cycle C. This is inconsistent with the two dual edges meeting in F. Thus v must have out-edges between e_1 and e_2.

Let $f_1, f_2, \ldots f_i$ be the set of out-edges from v which lie between e_1 and e_2 in clockwise order. Then let F_a be the face in G bordered by edges e_1 and f_1, and let a^* be the corresponding dual vertex. Similarly, let F_b be the face bordered by edges f_i and e_2, with b^* the corresponding dual vertex. The dual edges crossing $f_1, f_2, \ldots f_i$ form a dual

path M^* from a^* to b^*. By definition of RHOP, M^* is on the same RHOP c-tree as e_1^* and connects to it at a^*. Similarly, M^* is on the same RHIP c-tree as e_2^* and connects to it at b^*. Since by assumption both e_1^* and e_2^* are edges of cycle C found by the procedure *FindBoundaryCycle* and hence are overlap edges, all edges on M^* are overlap edges. Moreover, we know that since the edges of C were chosen from edges on the final trimmed RHOP and RHIP c-trees, all edges on the RHOP path from a^* to the root cycle must have been on the final trimmed RHOP c-tree. Similarly, the RHIP path from b^* to the root cycle must have been on the final trimmed RHIP c-tree. Thus if the orientation of the RHIP leaf edge f_1^* at a^* is positive, f_1^* must have been marked as good in step 5 and been included as a leaf in the final trimmed RHOP c-tree. Then in step 8, f_1^* must have been marked as a leaf of the trimmed RHIP c-tree which is a good leaf on the final RHOP c-tree. Thus the edges of M^* must be part of the final trimmed RHOP and RHIP c-trees.

We now show that the RHIP leaf orientation of f_1^* at a^* is positive. Look at the lo cycle C' defined by going forward along M^* and the RHIP path from b^* to either a lowest common ancestor of a^* and b^* in the c-tree or the root cycle, and then coming backwards on the RHIP path to a^*. By construction, the outside face is to the left of M^*, so that C' must be a clockwise cycle. Then by Property 4.6, the RHIP leaf orientation must be positive.

Hence, M^* must be a part of the subgraph considered in step 9 of the procedure, and M^* must either lie inside C or be a part of C. This means that v must be inside C, a contradiction to our original assumption.

4. We will now show that the set of cycles C found in the procedure *FindBoundaryCycles* obey the property that no boundary cycles of different sources share vertices.

The cycles found cannot intersect in an edge, since all edges in the cycles are overlap edges for the particular RHOP and RHIP c-trees belonging to the source. Suppose they intersected at a vertex. Then at that vertex there must be two in-edges belonging to different c-trees, and two out-edges belonging to different c-tree. However, by definition of RHIP and RHOP, all in-edges at a vertex must belong to the same RHIP c-tree and all out-edges must belong to the same RHOP c-tree. Hence the cycles found cannot cross at a vertex. ∎

4.3. Single source reachability and partitioning a planar dag

As we noted in the informal description, we need to consider each largest possible cluster of contiguous sources as one supersource in order to get the $O(\log k)$ bound for the number of iterations. In this section we will first give a procedure for finding the supersources. Once we find all supersources, we can then find boundary cycles for them and for the simple sources. These are in turn used to partition vertices according to which source or supersource can reach them. If we are only interested in finding all vertices reachable from a single source in a multiple-source dag, the problem of single source reachability, the two procedures, *FindSupersources* and *FindBoundaryCycles*, combine to give a simple algorithm. Partitioning out the vertices among all the multiple sources in a way consistent with depth first search requires more record keeping, although the basic step of using the boundary cycles to get smaller and smaller subproblems remains essentially the same.

4.3.1. Finding supersources

To find the supersources, we note that what we want is the outside boundary of the union of clockwise cycles which make up the positive faces. By Property 2.2, we can do this by finding the LHOP structure on the union of all the root cycles of the c-trees. Since in general, the union will not define a hole-free region, we need to check the orientations of the cycles to find the outside boundary.

procedure *FindSupersources*

(*Given the dual G^* of a planar dag with multiple sources, find the clockwise cycles enclosing the connected components of positive faces. We may exclude specified positive faces. Find, for each connected component of non-excluded positive faces, the sources which belong to the supersource. *)

1. Mark all edges of every positive face in G^*. If specified positive faces are to be excluded, the edges for such faces are not marked.
2. Look at the subgraph in the dual induced by the marked edges. Find the connected components of the subgraph, and label each component with edges from more than one positive face as a supersource. For every source in G whose associated dual positive face was not excluded, find to which supersource, if any, it belongs by finding to which connected component the edges of the associated dual positive face belongs.
3. Again look at the subgraph in the dual induced by the marked edges. Do cw vertex expansion at the complex dual vertices. These will be the vertices at which the cycles of the positive faces intersect.
4. Find the LHOP structure of the subgraph. Mark all the root cycles for the LHOP c-trees.
5. Run *CycleOrientation* to find the clockwise cycles out of the marked root cycles found in step 3. Mark these as the *outer* supersource cycles. Remove all edges and vertices introduced by the cw vertex expansion which are not actually on the marked LHOP root cycles.

Again all steps can be done in $O(\log n)$ time using $O(n)$ processors.

4.3.2. Single source reachability for a planar multisource dag

Algorithm 4. Single source reachability for a planar multisource dag

(* Given a planar dag G with multiple sources and a particular source r, find all vertices in G reachable from r. *)

While there exists a source besides r in the present G **do**

1. Run **procedure** *FindSupersources* to find all outer supersource cycles. Exclude the positive face surrounding r, so that r is not part of any supersource.
2. For every outer supersource cycle found in the previous step, replace all vertices inside the cycle by a new vertex which inherits all edges crossing the cycle and their embeddings. The new vertices will represent supersources in the enclosing graph. (As noted in Chapter 1, this is equivalent to contracting all vertices inside the cycle, and removing all self-loops.)

3. Run **procedure** *FindBoundaryCycle* to find all boundary cycles of all sources except for r.
4. Remove all edges of G which are crossed by the boundary cycles found in the previous step. (This is equivalent to replacing the boundary cycle in the dual graph by a single vertex.)
5. Run the algorithm for finding connected components of a planar undirected graph on G^u, the undirected version of G, and identify the component G_i^u containing r.
6. Identify the directed version of G_i^u as the new graph G.

Return the set of the vertices still remaining in G.

We note that since the graph at the end has only r as the source, all vertices in it must be reachable from r. Any vertex not in the graph at the end must have been inside some clockwise dual cycle with r outside, and hence could not have been reachable from r.

The number of processors needed in each step is linear in the size of G at that step so that the algorithm can be run using $O(n)$ processors. The time bound is $O(\log k \log n)$ since each step in the **while** loop can be done in $O(\log n)$ time and the number of iterations is $O(\log k)$ by the following theorem.

Theorem 4.6 *The algorithm for single source reachability will iterate at most* $\lceil \log k \rceil$ *times where* k *is the number of sources in* G *besides* r.

Proof. We will show that the total number of sources and supersources at the end of step 2 will be at least halved at each iteration. Since the boundary cycle for a source contains all the attendant vertices for that source, any new source created by the removal of edges crossing the boundary cycles must have had in-edges from at least two different boundary cycles. Since a boundary cycle is a simple clockwise cycle, the removal of the vertices inside a boundary cycle in step 4 must create a new face in G, and every new source which had an in-edge crossing that boundary cycle must be on the new face boundary. Thus all sources which a given boundary cycle contraction helped create must be contiguous. Thus one old source is part of at most one new supersource, and each new source or supersource must have been created by at least two old sources. Thus the number of sources and supersources is at least halved. ∎

4.3.3. Partitioning vertices in a multisource planar dag

In the algorithm for partitioning the vertices of a multisource planar dag among its sources, we cannot throw away parts of the graph as in the previous algorithm for single source reachability. At each iteration, we inherit a set of graphs, which are, with some modifications, the original graph partitioned into subgraphs. Each subgraph is then partitioned further into smaller subgraphs until every subgraph contains only one source. Every subgraph will be represented by the smallest boundary cycle which enclosed it.

In a given iteration, suppose we have a subgraph G_i containing sources other than its root source. We proceed as in *SingleSourceReachability* to find the boundary cycles for all sources and supersources in G_i except for the root source. Suppose we cut all edges of G_i crossing boundary cycles. We would then get one subgraph G_i' which is the same

as we would get if we ran one iteration step of the single source reachability algorithm on G_i, plus other subgraphs which are induced by vertices inside a boundary cycle. The subgraph G_i' is modified further by adding new sources formed by contracting vertices inside certain chosen boundary cycles. We note that if the source for a given boundary cycle is a supersource, the subgraph associated with the boundary cycle may contain vertices inside one of the marked counterclockwise LHOP root cycles found in procedure *FindSupersources*. This presents no problems since we will define all vertices belonging to a subgraph as those which are inside the associated boundary cycle and which are not inside any boundary cycle nested inside.

In order to keep track of the information needed for depth first search, we need to know which cycle is inside which cycle. Given a set of simple cycles, we will say cycle C_1 is *nested in* cycle C_2 if C_1 is inside C_2, and C_2 is inside every other cycle that C_1 is inside. We will use a *nesting tree* to keep a record of the nesting structure. This tree will have two types of nodes: source nodes and boundary cycle nodes. Both types of nodes may be active or inactive. Every boundary cycle node represents a boundary cycle and has as its children, one inactive source node, any number, including none, of active source nodes, and any number, including none, of boundary cycles nodes. The inactive child source node will represent the source of the boundary cycle, and the active children source nodes and boundary cycle nodes will be represent the sources and the boundary cycles which are nested in the parent boundary cycle. As noted above, each subgraph is represented by the smallest boundary cycle which enclosed it, so that each boundary cycle node is associated with a subgraph.

A source node will be one of four types: an atom source node which represents one of the original sources in the graph and is a leaf in the tree; a fake source node which is also a leaf and represents a new source which would be created if all edges crossing boundary cycles are removed; a simple source node, which has only one child, a boundary cycle node, and represents the contraction of vertices inside the boundary cycle; and a supersource node, which has two or more fake, simple, or atom source nodes as children and represents a supersource made from the children sources. (Figure 21)

In what follows, we will occasionally be lax about differentiating between a source or boundary cycle and the node in the nesting tree representing it. Thus we may talk about a source in the tree or a boundary cycle in the tree and mean the node in the tree corresponding to the given source or boundary cycle. We will also sometime use the notation $node(s)$ and $node(BC)$ to refer to the node representing source s and the node representing boundary cycle BC.

Whenever we have a boundary cycle node in the nesting tree with active source nodes as children, in the corresponding subgraph there will be more than one source. There will always be the root source for the subgraph (which was the source for the boundary cycle which enclosed the subgraph) represented by the inactive source node child of the boundary cycle node; any active source nodes represent extra sources which must be removed by contracting out their boundary cycles. The algorithm ends its iteration when every subgraph has only one source; in the nesting tree, this means no boundary cycle node has active source children.

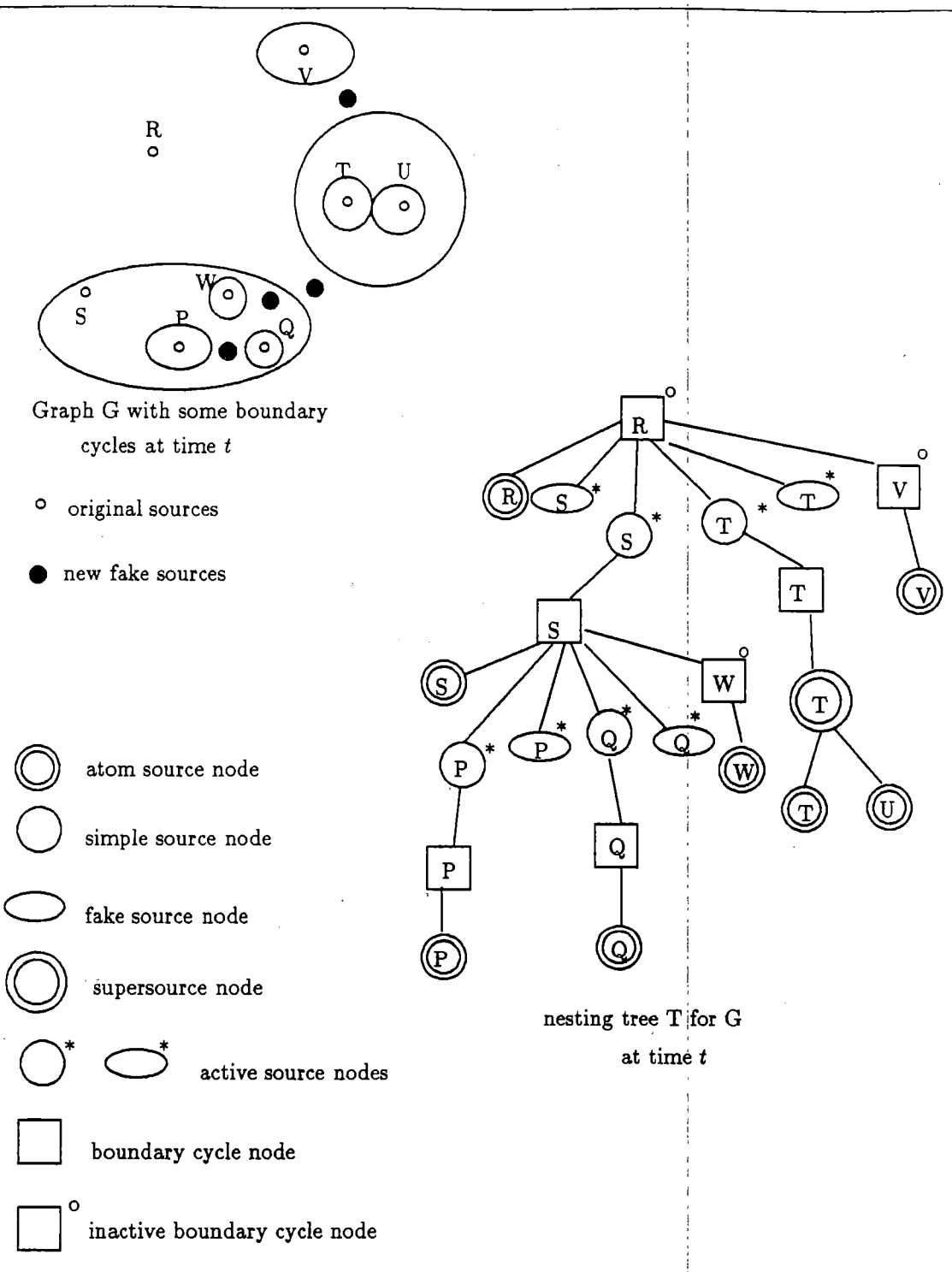


Figure 21
A nesting tree

When there are no more active children, if we then look at the path in the nesting tree up from an atom source leaf toward the root, we will in general find a path consisting of alternating inactive source nodes and boundary cycle nodes until we have one boundary cycle node as the parent of another. Since each subgraph associated with the boundary cycle nodes along the path has only one source, all vertices in the subgraph can be reached from that source. In turn, all edges crossing out of the boundary cycle form the edges of the only source for the subgraph in which it is nested. Thus as we go up the tree path of alternating source and boundary cycle nodes, all vertices in the subgraphs associated with the boundary cycles can be reached from the atom source which was the start of the path. If a boundary cycle node BC_1 is the parent of a boundary cycle node BC_2 on the tree path, then in the subgraph G_1 associated with the parent boundary cycle node BC_1, the subgraph G_2 associated BC_2 has been removed, and none of the vertices in G_1 will be assigned to the atom source at the start of the path. Thus if we cut the nesting tree at the edges between two boundary cycles, and each cut piece has only one atom source leaf in it, we can assign vertices belonging to subgraphs associated with boundary cycles in each piece to the atom source leaf.

This procedure needs modification when there are supersource nodes in the nesting tree, since this case can produce cut pieces with more than one atom source leaf. For example, suppose several atom sources created a supersource. Then all vertices in the subgraph associated with the boundary cycle for the supersource are reachable from the supersource, but a given vertex in the subgraph is not necessarily reachable from all atom sources making up the supersource. We will call a source node in the tree *processed* if all out-edges of the corresponding source have been assigned to atom sources. We will call a boundary cycle node *processed* if all vertices belonging to its associated subgraph has been assigned to atom sources. We will in the next section give a procedure which, given a source s with all its out-edges assigned to atom sources, will assign to atom sources all vertices in the subgraph with s as its only source. Using this procedure, we will process the cut pieces with more than one atom leaf, level by level, until all vertices in the subgraphs associated with the nodes in the cut pieces have been assigned.

Algorithm 5. Partitioning vertices among multisources

(*Assume we are given a planar dag G, its embedding, and a root source r. Consider as the outside face, the face bordering the root source r with the first edge out of r being a counterclockwise edge about it. We find a partition of the vertices and an ordering for the partition sets such that vertices in a lower numbered partition set have no out-edges to vertices in a higher numbered partition set.*)

1. Initialize:
   a. Find the dual G^*, do ccw vertex expansion for all complex vertices, and find the RHOP and RHIP structures of G^*.
   b. Build the nesting tree T consisting of the root boundary cycle node and all atom source nodes as children of the root boundary cycle node. Label the atom source node for r as inactive; label all other atom source nodes as active. (We can extend the definition of boundary cycles, and consider the boundary cycle for r to be the dual cycle created by doing a ccw vertex expansion of the dual

vertex associated with the outside face of G. If we designate the negative face formed in the dual as the outside face, then the entire graph G will be enclosed within this boundary cycle of r which is the boundary of the negative face.)

c. Place G in a set S of subgraphs. Label it $G_0(r)$ for consistency, where 0 represents the iteration step and r identifies the source for the boundary cycle which encloses the graph. Initialize I, the iteration step number, to 1.

2. (This is the main iterative step.) Given a set S of subgraphs $\{G_K(i)\}$'s for $K = 0 \ldots I-1$, their dual graphs, and the nesting tree T, while there exists a boundary cycle node in T which still has at least one active source child, then do for each $G_K(i)$ whose corresponding boundary cycle node in T has at least one active source child the following steps (a) through (g):

a. Run procedure *Findsupersources* excluding all the positive faces belonging to the source for the boundary cycle enclosing the subgraph. If the source for the boundary cycle was a supersource, we exclude all associated positive faces making up the connected component. Label every new supersource by the highest atom source number p from the sources it is composed of, and by the iteration number I as $SS(p, I)$. (This labelling scheme is fairly arbitrary; we just need some consistent way of naming the supersources created.) For every new supersource, update the nesting tree by putting in the supersource node as a new active child for the appropriate boundary cycle node with the sources composing it as its inactive children.

b. Do ccw expansion of any complex vertices in the dual.

c. Run *FindBoundaryCycle*, and find for each active source s its boundary cycle $BC(s)$ and label it. For this purpose only, temporarily contract all vertices inside an outer supersource cycle found in *FindSupersources* into a single vertex inheriting all edges crossing the cycle and their embeddings. Once the boundary cycle has been found, undo the temporary contraction. In T, make the new node for $BC(s)$ the new parent of s, which now becomes inactive. Mark all graph edges whose associated dual edge is part of a boundary cycle. (At this point the new nodes for the boundary cycles have no parents in T.)

d. Run procedure *CutGraph* given below to separate out subgraphs $\{G_{I+1}(j)\}$ at the boundary cycles from their parent graph $G_I(i)$, find their nesting relationships, and update the parent graph to produce $G_{I+1}(i)$.

e. Update the nesting tree to reflect the nesting relationships of the boundary cycles as follows. For every boundary cycle for which a new simple source in one of the $G_{I+1}(j)$ was created by *CutGraph*, make the corresponding boundary cycle node a child of a new active simple source node, which in turn will be a child of the boundary cycle node representing $G_{I+1}(j)$. For every vertex which became a new fake source in $G_{I+1}(j)$ because all its in-edges crossed boundary cycles, create an active fake source node as a leaf and make it a child of the boundary cycle node representing $G_{I+1}(j)$. For every boundary cycle nested inside $G_{I+1}(j)$ for which *CutGraph* did not create a new simple source, label the corresponding boundary cycle node as inactive, and make it the child of the boundary cycle

node representing $G_{I+1}(j)$. Label such inactive boundary cycle nodes in T with I, the iteration step at which they became inactive.

f. For every subgraph which has no active source, label all vertices in it as belonging to the source of the boundary cycle which represents it (i.e. the source of the smallest boundary cycle that enclosed the subgraph).

g. Increment I by 1.

3. In the nesting tree T, order the adjacency list for the links between a parent boundary cycle node and its children as follows:

   a. The source for the boundary cycle will always be first.

   b. All other children will be inactive boundary cycles. They will be ordered in the reverse order of the iteration step I at which they became inactive. Thus those which remained active longest will be put first after the source. If more than one boundary cycle child became inactive in the same iteration, any relative ordering can be adopted.

4. Find the preorder number of the inactive boundary cycle nodes in the nesting tree T, using the adjacency list ordering from the previous step. Note that simple source nodes have only one child, and the ordering among children of a supersource node is immaterial. Now cut T at each edge which connects an inactive boundary cycle node to its parent. This creates multiple trees from T. For every atom source leaf, find the inactive boundary cycle node which is the root of its tree. Let this boundary cycle node be the $rootBC(s)$ of source s. Give a partial order to the family of partition sets by the preorder numbering for $rootBC$'s found above: if a partition set for atom source s is the set of all vertices which will be in a tree with root s in an ET dfs forest, then the partial order number for the partition set for atom source s is the preorder number for $rootBC(s)$. Note that this is a partial order since all atom sources which have the same $rootBC$ will have the same partial order number. (Since every atom source has an associated partition set, we will sometimes refer to the ordering for the atom sources when rigorously we mean the ordering in the family of partition sets.) (Figure 22. Note that this is *not* the nesting tree in Figure 21 at a later stage.)

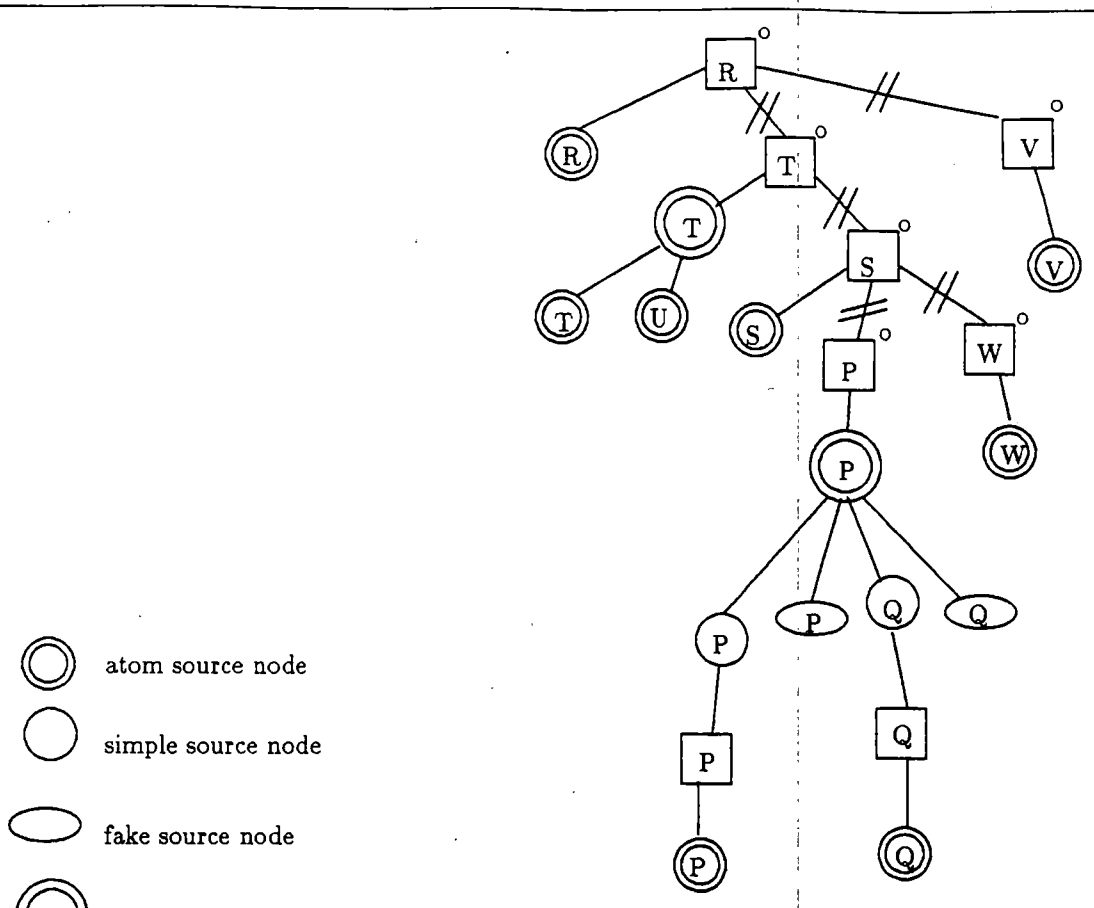
5. For every tree T' created from T in the previous step such that T' has only one atom source among its leaves, assign to the partition set for that atom source, all vertices labelled in step 2f as belonging to sources within T'. Also assign to the partition set for that atom source, any vertex which becomes a fake source in T'.

6. For each tree created in step 4 which contains more than one atom source, do the following:

   a. Process each atom node $node(s)$ by assigning all out-edges for the corresponding atom source s to that atom source s. (* This is a trivial initializing step.*)

   b. While the root boundary cycle node of the tree has not been processed, do the following steps (i) through (iv):

      i. For any fake source node, check if all in-edges to its corresponding fake source has been assigned, and if so, find the highest priority atom source s' out of all atom sources to which the in-edges have been assigned, and assign the out-edges of the fake source to s'.



atom source node

simple source node

fake source node

supersource node

boundary cycle node

inactive boundary cycle node

edge cut in step 4

Relative partition order of the atom sources

$$R < \{T,U\} < S < \{P,Q\} < W < V$$

Figure 22
A nesting tree at step 4 of Algorithm 5

ii. Process every supersource node for which all its children source nodes have been processed. (This step is also trivial since any out-edge of a supersource is also an out-edge for one of the sources which make up the supersource.)

iii. Process every boundary cycle node whose one source node child $node(s'')$ has been processed as follows: if all out-edges for s'' have been assigned to a single atom source s, assign to s all vertices labelled in step 2f as belonging to the source s''. Otherwise, run the procedure *AssignSupersourceVertices* given in the next section on the subgraph associated with the boundary cycle node.

iv. Process every simple source node whose boundary cycle node child has been processed. (Again, this is trivial since all out-edges of a simple source are out-edges from vertices which were assigned to atom sources when the child boundary cycle node was processed.)

c. Refine the partial order on the atom sources given by step 4 by ordering the atom sources in the same tree (i.e., the atom sources with the same $rootBC$) according to the priority system used by *AssignSupersourceVertices*.

The actual cutting of the graphs is done in the procedure below. Each step can be done in $O(\log n)$ time using $O(n)$ processors on a CRCW PRAM where n is the number of vertices in the graph.

procedure *CutGraph*

(* Given a graph G in which every edge whose dual is part of a boundary cycle for an active source is marked, we produce a set of subgraphs which partition the original graph G. *)

1. Cut G by removing all the edges marked as having an associated dual edge belonging to a boundary cycle. We will say these marked edges *cross* boundary cycles. For all vertices on either side of a cut edge, record the identity of the cut edge, which boundary cycle the cut edge crossed, and whether the vertices are inside or outside that boundary cycle.
2. Find all the connected components, using the algorithm for undirected graphs.
3. For each component, make a list of boundary cycles which were nested inside. For this step we make use of the information from steps 1 and 2, by which each vertex on either side of the cut edge knows to which connected component it belongs.
4. For every component which had a nested boundary cycle, do the following:
   a. Find all new sources. For each new source, find the highest priority boundary cycle which had been adjacent to it, and label the cut edge coming from it as *chosen*. Label the new sources as fake sources.
   b. For every boundary cycle, check to see if any cut out-edge which used to cross it was labelled *chosen* by a new fake source. If for any boundary cycle, there exist no such chosen out-edges, label the boundary cycle as inactive.
   c. If a boundary cycle has at least one chosen out-edge crossing it, create a new vertex v in the graph component in which it is nested. The new vertex v inherits all cut out-edges belonging to the boundary cycle, except for those out-edges that pointed to the new fake sources. Label these new vertices as simple sources.
   d. For each new simple source vertex v created in the previous step, remove all multi-edges and self-loops.
5. Return the set of subgraphs which are the modified connected components.

In the remainder of this section, we show that the partitioning algorithm produces the correct depth first ordering of sources and satisfies the time and processor bounds assuming that procedure *AssignSupersourceVertices* can be done on a graph with n vertices in $O(\log n)$ time using $O(n)$ processors. The assignment of vertices inside a boundary cycle for a supersource to sources composing the supersource will be covered in the next section.

Theorem 4.7 *A vertex w in G which has been assigned to an atom source s_1 in step 5 of Algorithm 5 will be reachable from s_1. Let u be in a tree created in step 4 with a root $rootBC(s_1)$, and v be in a tree with root $rootBC(s_2)$ where the two roots and hence the two trees are different. Then if $rootBC(s_1)$ has a lower preorder number than $rootBC(s_2)$, u will have no directed path to v.*

Proof. Consider T in step 4 in Algorithm 5 after the preorder numbers have been found (Figure 22). Every simple source node has just one child, a boundary cycle. Supersource nodes have two or more simple, fake, or atom sources as children. Fake and atom source nodes have no children. A boundary cycle node always has one source node and can have one or more inactive boundary cycle nodes as children. Note that by step 2e, any boundary cycle node which was a child of another boundary cycle node becomes inactive or becomes the child of an active source. At the end of the iterations of step 2, no boundary cycle node has active sources as children. Thus when T is cut in step 4, every tree which is created has an inactive boundary cycle node as the root and will either be a linked list of alternating boundary cycle and simple source nodes with an atom source at the leaf end, or a tree whose only branching comes from the supersource nodes. We note that a supersource node can never have only fake source nodes as children, because when fake source nodes are created in step 2e of the partitioning algorithm, at least one simple source node must have been created as a sibling, representing the contracted neighboring boundary cycle crossed by the marked cut edge as found in step 4 of *CutGraph*.

We first show that a vertex w which was assigned to an atom source s_1 in step 5 of the partitioning algorithm will be reachable from s_1. Let T' be a tree created from T in step 4 which has the node $node(s_1)$ representing s_1 as the only atom source leaf. Suppose T' has supersource nodes with branchings. A supersource node in T' can only have one simple source node as a child, since the presence of more than one simple source node would imply more than one atom source leaf.

By step 2f of the partitioning algorithm, a vertex w assigned to a source s is in a dag with just one source s and hence is reachable from it, since all other sources have been contracted out inside their boundary cycles. There are three possibilities for s: it may be the atom source s_1, it may be a simple source, or it may be a supersource. We note that it cannot be a fake source since all fake sources will become part of a supersource in the next iterative step, and thus no boundary cycle can have a fake source as its source.

Suppose s is not s_1, but is represented by some simple source node $node(s)$ in T'. Then every out-edge of s is actually an out-edge crossing a boundary cycle BC, where $node(BC)$ is the boundary cycle node child of $node(s)$ in T'. Since every out-edge crossing BC is reachable from the the source s', where $node(s')$ is the inactive source node child of $node(BC)$, w is reachable from s'. We have gone two levels down in T', and if s' is still not the atom source s_1, we can recurse.

If s was a supersource instead of a simple source, we can still use the same reasoning, since every supersource node has only one simple node as its child. Thus if supersource node $node(s)$ has $node(BC)$ as its grandchild, then every out-edge of s is either an out-edge crossing the boundary cycle BC or is an out-edge of a vertex represented by a fake source node which in turn is reachable by an out-edge crossing BC. Thus every vertex reachable from s is again reachable from the source of BC, now three levels down in T', and we can recurse until we reach the atom source node $node(s_1)$.

We will now prove the second assertion in the theorem. Look at the relationship of u and v. We note that if for any subgraph, we find that there exists a clockwise cycle in the dual such that u is outside and v is inside, this means that there is no path from u to v, since all edges of the subgraph crossing the cycle point out.

Let $bc(u)$ denote the boundary cycle node whose associated subgraph is the one in which u is found at the end of the procedure, and similarly for $bc(v)$. Thus $bc(u)$ is in the tree for s_1 with $rootBC(s_1)$ as root; the same relationship holds for $bc(v)$ and $rootBC(s_2)$. If the preorder number of $rootBC(s_1)$ is less than for $rootBC(s_2)$, then $rootBC(s_1)$ is either an ancestor of $rootBC(s_2)$ in the nesting tree, or the two have a lowest common ancestor and $rootBC(s_1)$ is visited before that of $rootBC(s_2)$ in the dfs of the nesting tree.

Suppose the ancestor relationship holds. Both $bc(u)$ and $bc(v)$ are then in the subtree of T (before the cutting of T in step 4) which has $rootBC(s_1)$ as the root, and $bc(u)$ will be on the tree path from s_1 to $rootBC(s_1)$, which alternates source node and boundary cycle node as noted above. Find the lowest common ancestor X of $bc(u)$ and $bc(v)$. X will also be on the tree path from s_1 to $rootBC(s_1)$. In the subgraph associated with this boundary cycle node X, the boundary cycle represented by $bc(u)$ will either be the source or nested inside the source, while $bc(v)$ will be either an inactive boundary cycle node in the graph or nested inside one of the inactive boundary cycles. There will thus be a clockwise cycle enclosing v but not u, and there can be no path from u to v.

A similar analysis holds for the case when $rootBC(s_1)$ and $rootBC(s_2)$ are not in ancestor-descendant relationship. Find the lowest common ancestor of the two nodes in T. Let this be the boundary cycle node Y; Y will also be the lowest common ancestor of $bc(u)$ and $bc(v)$. The paths to $bc(u)$ and $bc(v)$ start with different children of Y. If the path to $bc(u)$ starts by going to a source child, the case is the same as above. If the two start by going to inactive boundary cycle nodes, the adjacency list ordering of step 3 indicates that either the one which is an ancestor to $bc(v)$ was inactivated earlier or in the same iteration as that for the ancestor to $bc(u)$. If it was in the same iteration, this means there was then two non-nested boundary cycles one containing u inside and the other v. In the other possibility, at the time the boundary cycle node which is the ancestor of $bc(v)$ became inactive, the subgraph had a boundary cycle containing v, while u was outside and may still have been unassociated with any boundary cycle. In either case, there exists a clockwise cycle enclosing v, but not u, and hence there cannot be a path from u to v. ∎

Theorem 4.8 *Algorithm 5 for partitioning a multi-source planar dag will run in* $O(\log k \log n + T(n) \log k)$ *time and use* $O(n + P(n))$ *processors on a CRCW*

PRAM, where $T(n)$ and $P(n)$ are the time and processor bounds for the procedure AssignSupersourceVertices and where k is the number of sources in the dag.
Proof.

1. We show that the size of the problem never grows to be more than $O(n)$. First, we see that due to an argument similar to one used in finding the time bound for Single Source Reachability, the total number of active sources at the end of step 2a will always be at most half that in the previous iteration. Every active source at the end of step 2a will be represented by a boundary cycle at the end of 2b. All new fake sources which had an in-edge crossing a given boundary cycle will be in the same cluster of contiguous sources, since either the boundary cycle became inactive, in which case all the fake sources would be on the same face, or a new simple source was created, in which case all fake sources will be contiguous to the new simple source. As before, every fake source, and hence every supersource must have at least two boundary cycles, contributing to it. We note that in this algorithm, all new active sources at the end of step 2e will become part of a supersource at the end of step 2a in the next iteration, Thus, every boundary cycle contributes to only supersource, and every supersource must have at least two boundary cycles contributing to it. Since a boundary cycle represents an active source from the previous iteration, the number of active sources is at least halved. Thus the total number of boundary cycle nodes will be $O(k)$, where k is the number of sources originally present in G (i.e. k is the number of atom sources). The total number of simple source nodes will never be more than the total number of boundary cycles found and will by the above argument be $O(k)$. The total number of fake source nodes can never be more than n, the number of the original vertices in G. Thus the total number of nodes in T at the beginning of step 3 will be $O(n+k)$, and hence $O(n)$.

For all steps dealing with the set of subgraphs, the number of processors is dependent on the total number of vertices in the subgraphs. (Note that since these graphs are planar, and G and all subgraphs created from it have no multi-edges or self-loops, the number of edges and faces are linear in the number of vertices in the graph.) We will therefore show that the number of vertices for all subgraphs at the start of each iteration will never be more than $O(n+k)$, i.e. $O(n)$.

Look at the steps where new vertices are added to G or to one of the subgraphs later created. In the initialization step, ccw vertex expansions are done in the dual G^*; this means that sinks are added to certain faces of G. There can never be more that $O(n)$ such new sink vertices added, so at the start of step 2, the number of vertices in the graph is still $O(n)$. Inside the iterative step, step 2b again adds sinks to the subgraphs when it does vertex expansion. The number is bounded by the number of new faces which were created in *CutGraph* in the previous iteration. There are three ways new faces are created in *CutGraph*. First, when vertices inside an inactive boundary cycle are removed, they will leave a new face in the subgraph in which the boundary cycle was nested. Second, in step 4c, when a new active simple source is created all out-edges are kept except those going to new fake sources which did not restore and mark them. These edges are removed, creating new faces. However, these faces will be those in which at least two active sources are on the boundary by construction. Thus in step 2a in the partitioning algorithm, these faces will be destroyed in the creation of the supersources, before the vertex expansion

step. The third type of new faces are the outside faces for the subgraphs which were created by cutting all out-edges crossing the boundary cycles in step 1 of *CutGraph*. Thus the total number of faces which could be involved in the vertex expansion of the duals is linear in the number of boundary cycles. Since this is $O(k)$, the total number of vertices added by step 2a in the partitioning algorithm is also $O(k)$.

Vertices are also created in the procedure *CutGraph*. The fake sources are not additions, since they are vertices which were originally in the subgraph which became sources when all their in-edges were removed. The simple sources are created in step 4c, but their total number cannot exceed the number of boundary cycles found, and hence is $O(k)$.

Thus the total number of vertices in the set of subgraphs is $O(n+k)$. Since each step in the algorithm with the exception of step 6, can be done with a number of processors linear in the number of vertices in the set of subgraphs involved, the entire algorithm can be run with $O(n + P(n))$ processors, where the procedure *AssignSupersourceVertices* uses $P(n)$ processors.

2. We now show that the algorithm runs in $O(\log k \log n + T(n) \log k)$ time on a CRCW PRAM, where the procedure *AssignSupersourceVertices* runs in $T(n)$ time. Step 1, the substeps of step 2, and steps 3 through 5 can all be done in $O(\log n)$ time on a CRCW PRAM. The number of iterations of step 2 is $O(\log k)$, by the argument given above in part 1 that the number of active sources must be at least halved at each step.

We now show that the height of any tree created from T in step 4 of the partitioning algorithm with more than one atom source leaf is $O(\log k)$. As noted in the proof of Theorem 4.7, such a tree consists of alternating levels of boundary cycle nodes and either simple source nodes or simple source nodes linked to a parent supersource node. The boundary cycle nodes in such a tree which are in ancestor-descendant relationship cannot have been created at the same iterative step. We show this by contradiction.

Assume BC_1 and BC_2 are two boundary cycle nodes in the same subtree, such that BC_2 is a descendant of BC_1 and both were found in the same iteration at step 2c. Then the nodes representing their sources must have been active at the beginning of that step 2c, and have become the inactive child of the new boundary cycle nodes at the end of step 2c. If these sources were children of different boundary cycle nodes in T, they will never be nested one inside the other, so we assume they were both children of a common boundary cycle node. We then conclude that BC_2 was found to be a descendent of BC_1 in the nesting relationship found in step 2d; else the two will never be in the proper ancestor-descendent relationship in T at the end. However, at the end of step 2c, BC_1 already has as its child its inactive source node, and any boundary cycle node found nested inside BC_1 in step 2d will eventually be a descendant of an inactive boundary cycle node of BC_1. Thus when the nesting tree is cut in step 4, BC_1 and BC_2 will end up in different trees. This contradicts our assumption.

Since there are at most $O(\log k)$ iterations, the height of any tree created from T in step 4 must be $O(\log k)$. Thus the time for step 6 must be $(T(n) \log k)$, and the entire algorithm has time bound $O(\log k \log n + T(n) \log k)$. ∎

4.4. Assigning vertices inside supersource boundary cycles

Suppose we are given a planar dag in which all the sources are part of one cluster of contiguous sources. We will show in this section how to assign a vertex in the graph to a specific source. (In this section only, a vertex will mean a non-source vertex in the graph.) We will assume that the sources are ordered by some priority system; we will explain how the priority numbers are assigned more fully below. We will call this number $priority(s)$ for the source s. The procedure *AssignSupersourceVertices* will assign a vertex to a source with the highest priority number which has a path to the vertex.

We will first explain the procedure to cut the graph into components and to reduce the problem to assigning vertices to sources in a graph which has all the sources in a line bordering the outside face. We then once more discuss properties of the RHOP and RHIP structures, and how they can aid in finding the set of sources which have paths to a vertex. The main idea is that there will always be a path from a source to a vertex in the graph unless a back arc encloses the source but not the vertex, or a forward arc encloses the vertex but not the source.

Suppose we call the set of sources which have a path to a vertex v the set $Possiblesource(v)$. We need to assign v to the source in $Possiblesource(v)$ with the highest priority. To find the sources in $Possiblesource(v)$, we look at the endpoints of the smallest forward arc which encloses a vertex. We can eliminate all sources lying to the right of the right endpoint or lying to the left of the left endpoint from the possibility of being in $Possiblesource(v)$. If we can then find all the largest back arcs which do not enclose v, but are inside the smallest forward arc, we can eliminate the sources lying inside the back arcs from the possibility of being in $Possiblesource(v)$. The sources left are the sources of $Possiblesource(v)$.

By preprocessing the graph, we create a type of range tree such that, given a vertex v and the smallest forward and back arcs enclosing it, one processor in $O(\log n)$ time can find the highest priority source in $Possiblesource(v)$. We do this by cutting edges in G which cross the back arcs. Each separate component will be called a *hole*. We will show that the holes can be organized into a nesting tree, such that if a vertex belongs to a hole, it will be enclosed by the back arcs associated with the holes which are its ancestors in the tree, and by no other back arcs. Further preprocessing using the tree will give the data structure needed.

Once the graph is cut into subgraphs as explained below, the problem is basically the same as assigning vertices to different out-edges of a single source, and it might seem that we can simply use the single source dfs algorithm from Chapter 3, and adjust the priority ordering to correspond. Unfortunately, this will not work since the ET dfs gives a right to left or left to right priority ordering of the out-edges. In the cyclic ordering of the out-edges of a supersource along one of the cycles marked by *FindSupersources*, all the out-edges of a given atom source will in general not be in one consecutive group. Thus we need an algorithm which will assign vertices according to a global priority order, regardless of the ordering of the out-edges.

Although steps 1 through 4 of the partitioning algorithm fix a partial order among the partition sets for the atom sources, they impose no constraint on the relative ordering of partition sets for atom sources which have the same *rootBC* in the nesting tree. We

should perhaps note at this point that although in the procedure *CutGraph*, a new fake source chooses the highest priority boundary cycle adjacent to it and an out-edge crossing it, the procedure *AssignSupersourceVertices* may end up assigning the vertex which becomes the fake source to an atom source which may not even be inside the highest priority boundary cycle. As long as a boundary cycle which has an edge crossing it incident on the new fake source does not become inactive, it will become a sibling simple source along with the highest priority boundary cycle with the edge marked by the fake source. Basically, the reason for having a fake source choose is to make some non-higher priority boundary cycles inactive. If the boundary cycles become new simple sources, there is is no future distinction among them with respect to the fake source; the new simple sources all become part of a supersource along with the fake source.

We are therefore free to impose any priority order we wish within the set of the atom sources which have the same $rootBC$. As we process a tree with more than one atom source leaf in step 6 of the partitioning algorithm, at a given time, one of the sources composing the supersource may have out-edges assigned to more than one atom source. The procedure *CutSupersource* given below deals with this by creating a set of duplicate atom sources, one for each consecutive set of out-edges assigned to the same atom source crossing an LHOP root cycle. A duplicate of an atom source s will have the same $priority(s)$ number; the presence of several sources in the supersource with the same priority number will not create any problems.

4.4.1. Cutting the supersource graph

The procedure *FindSupersources* given in the previous section finds all LHOP root cycles which, by Property 2.2, are the cycles in the dual graph enclosing the sources in G making up the supersource. In general the region composed of the positive faces of a cluster of contiguous sources will not be hole-free, and there will exist counterclockwise cycles among the LHOP root cycles which define inner regions with vertices which must be assigned. The procedure *CutSupersource* will cut the graph into separate subgraphs using these cycles, one subgraph for each cycle, creating copies of the sources for each subgraph as needed. We will then use the single source ET dfs algorithm from Chapter 3 to find the vertices reachable from the source with highest priority in each subgraph. By removing this source and its vertices, we cut open the cycle; we then define as an outside region, the face created by the removal. This creates an embedded planar graph in which all the sources are on the outside face. By defining the sources to be at the bottom, we give a left to right ordering to all the sources remaining. This will simplify the descriptions of procedures on the graph.

procedure *CutSupersource*

(* Given a planar dag G all of whose sources are part of one cluster of contiguous sources, this procedure produces a set of subgraphs, which partition the graph in such a way that for each subgraph, all sources are in one outside face. Given assignments to atom sources for all out-edges of the sources, the sources in each subgraph will be duplicates of atom sources with all out-edges for a given duplicate atom source in one consecutive group in the ordering of out-edges crossing the one outside face. *)

(Figure 23)

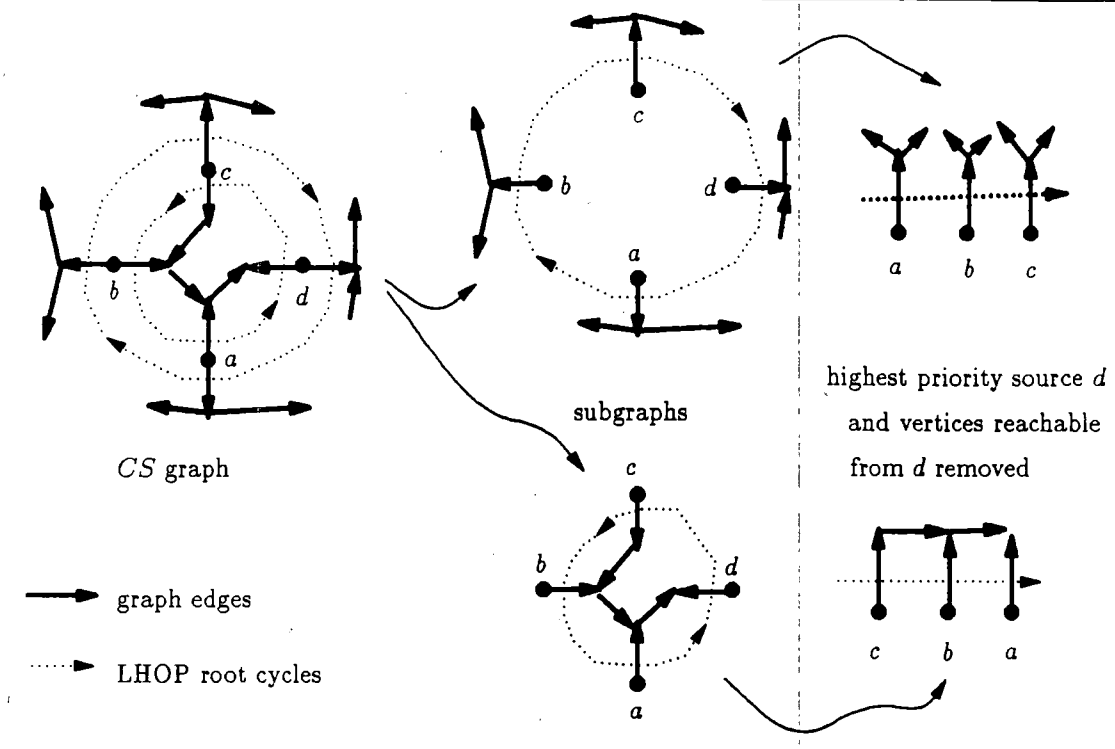


Figure 23
Cutting a graph with a supersource as the only source

1. Run procedure *FindSupervertices*, keeping all LHOP root cycles found in step 3, not only the clockwise one.
2. For every LHOP root cycle found, make a clockwise ordered list of all out-edges of sources crossing the cycle.
3. For each list of out-edges from step 2, find the groups of consecutive out-edges belonging to the same atom source. Remove all the old sources in the graph and replace them with new sources, one for each group of consecutive out-edges just found. Give each new source as its only out-edges the associated group of consecutive out-edges, and set the priority of the new source to be the same as that of the atom source to which its out-edges were assigned. Note that the new sources will have out-edges crossing only one particular LHOP root cycle. Since the LHOP root cycles partitioned the non-source vertices into separate components connected only through the old sources, this effectively separates the original graph into subgraphs cut apart at the sources. For each subgraph the original LHOP root cycle will define a region to the right of the cycle which has only sources in it.
4. Find the highest priority source in each subgraph. If there is more than one, choose one arbitrarily. Starting at the rightmost out-edge of that source (i.e., the last edge in the group in a clockwise ordering about the LHOP root cycle), run Algorithm 3 and do a right-handed planar Euler tour depth first search for each

subgraph, considering all out-edges from the sources to be coming from one source. This will find all vertices reachable by the highest priority source.

5. Remove the highest priority sources and the vertices reachable from them found in the previous step. This leaves subgraphs whose sources are all on the face created by the removal of the source and its vertices. Label as LeftSource the source immediately after the removed source in the clockwise cycle, and label the source before it as RightSource. We will now orient the graph so that the out-edges of the sources point up and their left and right orientations are consistent with the LeftSource and RightSource designation.

We note that the number of new sources cannot exceed the number of out-edges for the supersource. Hence the total number of vertices in all subgraphs considered together will still be $O(n)$. Each step of the procedure can thus be done in $O(\log n)$ time with $O(n)$ procedures on a CRCW PRAM, and the entire procedure thus has the same bounds.

4.4.2. More properties of RHOP and RHIP structure

For the remainder of this section on assigning vertices within a graph whose only source is a supersource, we will always assume that we are given a subgraph which is the output of procedure *CutSupersource*. We will call such a subgraph a CS graph. Although technically the dual of a CS graph has a set of connected positive faces, all connected together at a single dual vertex, we will for the dual structure only, assume that all sources in a CS graph are contracted into one source. Then the root cycle will correspond to the LHOP root cycle used to cut up the original graph into the given CS subgraph. Thus a CS graph is a special graph in which all sources are in the outside face of the dual and inside the same root cycle, with the sources ordered from left to right. In this restricted case, we can find further properties of the RHOP and RHIP structures to help us. We further assume that all dual vertices are simple; this will be the case for all CS graphs produced when running the partition algorithm.

We note first that all dual edges can now be considered overlap edges, since there is only one root cycle in each subgraph. We can thus employ the classification given in Property 4.3 to classify the pair of RHOP and RHIP paths associated with every dual edge.

Property 4.8 *Every dual edge e^* in a CS graph must be associated with a pair of RHOP and RHIP paths in one of the following classes: a forward arc, a back arc, one or more back twists followed by a forward arc, or one or more back twists followed by a back arc.*

Proof Sketch. The RHOP path from the root cycle to edge e^* and the RHIP path from e^* back to the root cycle can intersect only in a back twist in the subgraphs produced by procedure *CutSupersource*. A forward twist is impossible since the sources are in the outside face, and there cannot exist a cycle to separate the root cycle and the outside face. ∎

In the remaining discussion, we will call edges which define forward arcs as *forward* edges, edges which define back arcs as *back* edges, and edges which define back twists ending in back or forward arcs as *bt* edges.

Property 4.9 *In a CS graph we can distinguish whether a pair of RHOP and RHIP paths associated with a dual edge ends in a back arc or a forward arc by the following: if the endpoint of the RHOP path is to the left of the RHIP path, it is a forward arc; if it is the the right, it is a back arc.*

Similarly, in the classification of the RHIP leaf orientations, the cases in which a cycle separates the root cycle from the outside face are not possible, thus making the following properties true.

Property 4.10 *Only cases 1A and 3, and negative case 2 are possible cases of RHIP leaf orientations in a CS graph.*

Property 4.11 *In a CS graph, if we assign preorder numbers to all vertices and edges in the RHIP c-tree by using a righthand Euler tour starting at the RightSource on the root cycle, then for all negative leaf orientations,*

$$preorder(u) < preorder(e).$$

For all positive leaf orientations,

$$preorder(e) < preorder(u).$$

Let P' be a RHOP path in a CS graph from the root cycle to a leaf in the RHOP c-tree. The following properties will help classify the RHOP-RHIP path pairs for the edges on P'. In the following, we will let e_1 be an edge on P', and P_1 and Q_1 be the RHOP path and the RHIP path respectively from e_1 to the root cycle. Let e_2 be an edge on P' between e_1 and the root cycle (i.e., it will be on P_1), with P_2 and Q_2 the RHOP and RHIP paths from e_2 to the root cycle. Let $RHOProot_1$ be the endpoint of P_1 on the root cycle, and $RHIProot_1$ be the endpoint of Q_1 on the root cycle. $RHOProot_2$ and $RHIProot_2$ are defined similarly. Note that $RHOProot_1$ and $RHOProot_2$ and the endpoint of P' are all the same.

Property 4.12 *If P_1 and Q_1 form a forward arc, P_2 and Q_2 will also form a forward arc. $RHIProot_2$ either is the same as $RHIProot_1$ or lies to its left. (Figure 24)*

Proof Sketch. By definition of the RHOP and RHIP structures, a RHIP path can leave a RHOP path only on the right. Thus Q_2 must lie on or in the cycle formed by the forward arc of P_1 and Q_1 and the root cycle between their endpoints. It cannot get outside because it cannot cross a RHOP path from the right or cross any RHIP path. ∎

Property 4.13 *If both pairs of RHOP-RHIP paths form back arcs, the $RHIProot_1$ either is the same as $RHIProot_2$ or lies to its right. (Figure 25)*

Proof Sketch. Again, because an RHIP path can diverge from an RHOP path only to the right of the RHOP path, and an RHIP path cannot cross an RHOP path from the right to the left, the section of P' from e_2 to the leaf is inside the cycle created by P_2, Q_2, and the section of the root cycle between their endpoints. Specifically, e_1 will be inside, and Q_1 will not be able to cross Q_2, so that its endpoint must be on the endpoint of Q_2 or to its right. (Note if Q_1 crossed P_2, e_1 would no longer define a back arc.) ∎

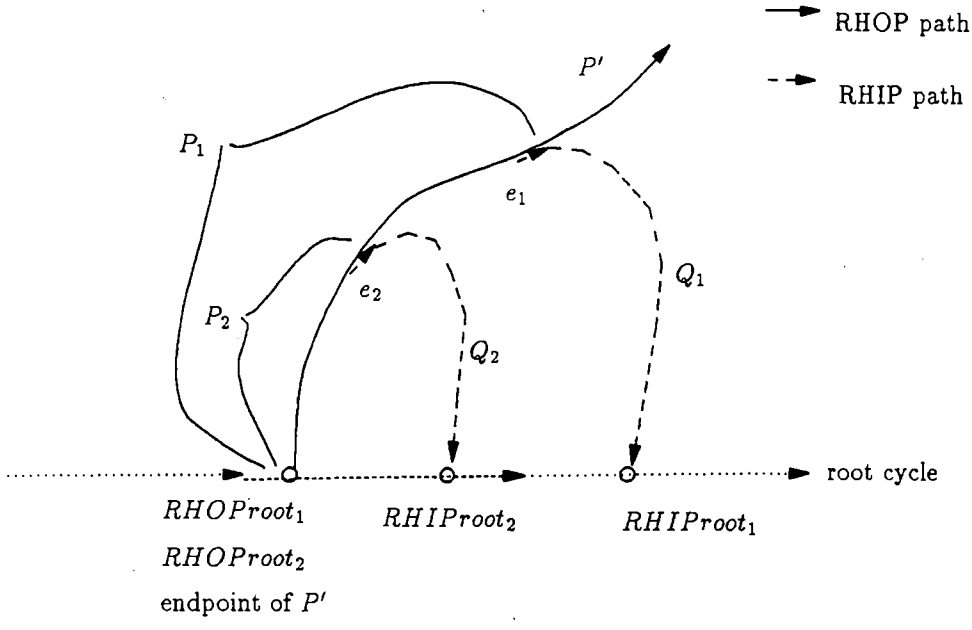


Figure 24
Illustration of Property 4.12

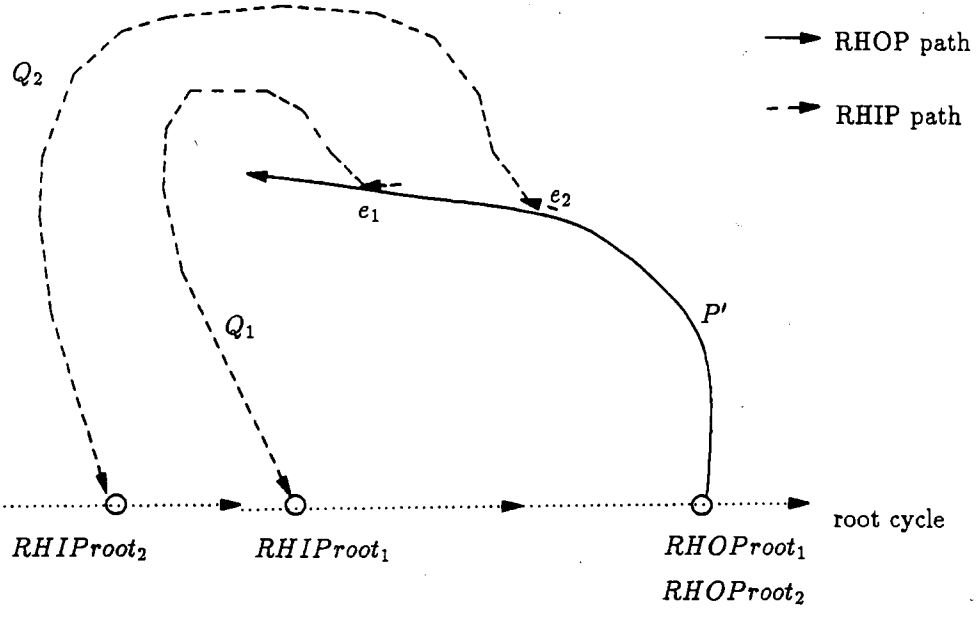


Figure 25
Illustration of Property 4.13

Property 4.14 *If P_2 and Q_2 intersect in a back twist, so must P_1 and Q_1. (Figure 26)*

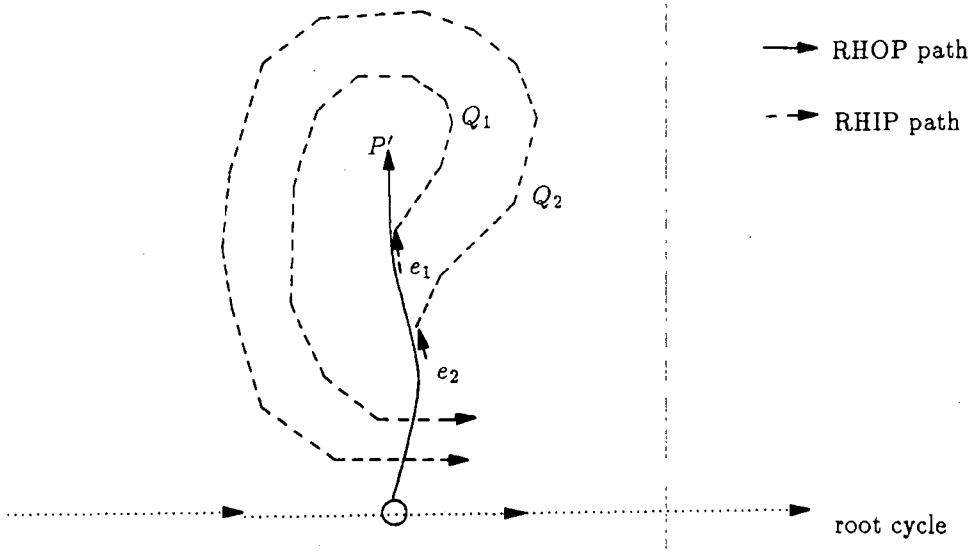


Figure 26
Illustration of Property 4.14

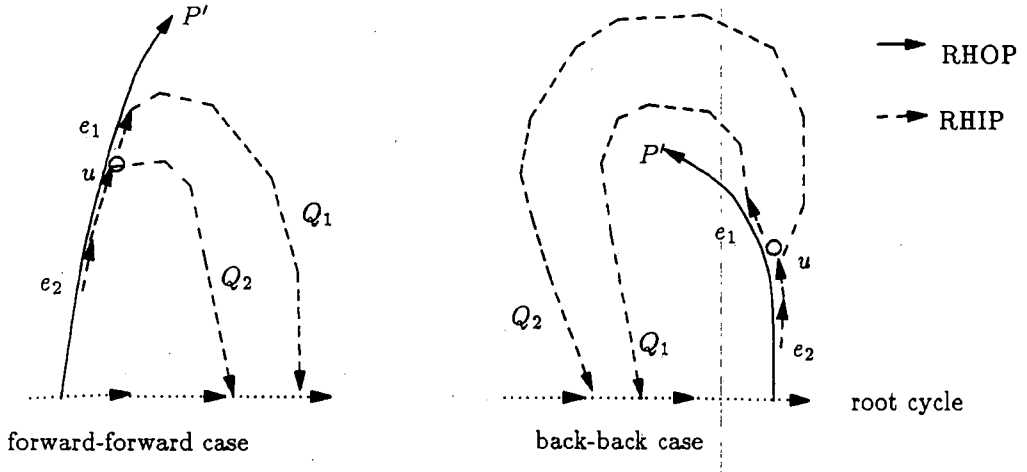


Figure 27
Illustration of Property 4.15

Proof Sketch. The only way that Q_1 can exit the back twist cycle formed by the intersection of P_2 and Q_2 is to cross P_2. This means that it will intersect P_1 since P_2 is part of P_1. ∎

Property 4.15 *If e_1 and e_2 satisfy the further requirement that $e_1 = (u, v)$ is a leaf in the RHIP c-tree and u is a vertex on the RHIP path Q_2, and if both e_1 and e_2 are forward edges, then the leaf orientation of e_1 is positive. If both e_1 and e_2 are back edges, the leaf orientation of e_1 is again positive. (Figure 27)*

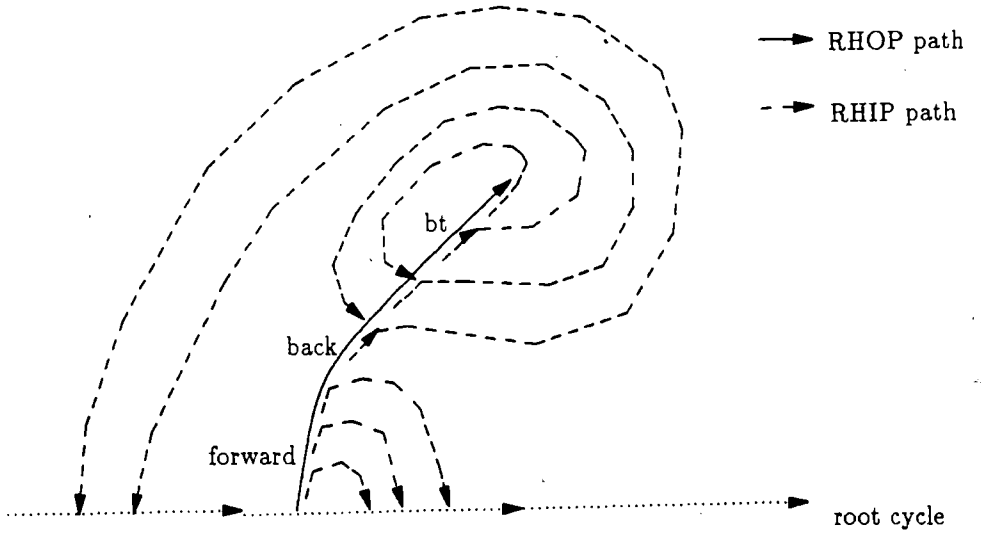


Figure 28
Types of edges on P'

Proof Sketch. By the above properties 4.12 and 4.13, $RHIProot_1$ must be the same as $RHIProot_2$ or lie to its right. If the two path endpoints are the same, the branches of the c-subtree containing e_1 and e_2 must have relative orientations so that $preorder(e_1) < preorder(u)$. ∎

The above properties allow us to quickly distinguish between the three types of edges on a RHOP c-subtree. Assuming that all three types of edges are present on P', starting at the root cycle, they must be ordered as follows: the forward edges are followed by back edges, which in turn are followed by bt edges. P' may have no edges of a given type, but if it has a bt edge whose associated RHOP-RHIP paths end in a back arc, there must be at least one back edge on P'. Similarly, if there exists a bt edge on P' whose associated RHOP-RHIP paths end in a forward arc, there must exist at least one forward edge on P'. (Figure 28)

If P' has any back edges, the back edge closest to the root cycle is easily found by looking at the endpoints of the associated RHOP and RHIP paths and using Property 4.9. We will call this back edge a *RHOPfirstback* edge.

Property 4.16 *Let the dual edge g^* be the first bt edge on a RHOP path P from the root cycle to a leaf. Then if all edges before g^* on P were forward edges (this immediately implies that g^* defines back twists ending in a forward arc), g^* has a negative leaf orientation in the RHIP c-tree. Moreover, it will be the first negative leaf orientation when looking at all RHIP leaf orientations for dual edges from the root cycle to g^* along P. If g^* is a bt edge defining back twists ending in a back arc (this immediately implies that there are back edges on P before g^*), then g^* will have the first negative RHIP leaf orientation found when looking at RHIP leaf orientations for dual edges on P after the RHOPfirstback edge.*

Proof Sketch. By Theorem 4.1, we know there must exist one negative leaf orientation for a dual edge along P between g^* and the dual vertex where the RHIP path crosses P to give a back twist. By Property 4.15, we know this must be at g^*. ∎

By using the previous properties, we can give a procedure for labelling all edges in a RHOP c-subtree by their types. Each step in the following procedure can be done in $O(\log n)$ time using $O(n)$ processors on a CRCW PRAM.

procedure *LabelForwardBack*

(* This procedure takes a planar subgraph G and its dual G^* with their embeddings, which are the results of procedure *CutSupersource*, and labels dual edges as forward if they define forward arcs and back if they define back arcs. The back edge which has no other back edge on the path between it and the c-root of its RHOP or RHIP c-subtree gets a special label *RHOPfirstback* or *RHIPfirstback*. *)

1. Find the RHOP and RHIP structures of the dual G^*. For each edge in each c-subtree, find the c-root of the c-subtree it is in. (Remember that there is only one root cycle enclosing all sources.)
2. Using LeftSource and RightSource orientation of the sources from procedure *CutSupersource*, determine the left to right ordering of all c-roots on the root cycle.
3. Do a righthand Euler tour of the RHIP c-tree, starting at the rightmost RHIP c-subtree, and assign preorder numbers to all edges and vertices in the RHIP c-tree.
4. For every dual edge, label it *maybeforward* if its RHIP c-root is the the right of its RHOP c-root. Otherwise, label it *maybeback*. (Both *maybeforward* and *maybeback* edges may include bt edges.)
5. For every RHOP c-subtree, by pointer doubling, find the *maybeback* edges which have no other *maybeback* edges as their ancestors. Label these *RHOPfirstback*.
6. Cut the RHOP c-subtree branches at the *RHOPfirstback* edges, so that the exclusively trimmed RHOP c-subtrees consist only of *maybeforward* edges.
7. For every edge in the exclusively trimmed RHOP c-subtrees which is also a leaf in the RHIP c-tree, find the RHIP leaf orientation by using the preorder numbers found in step 3.
8. Exclusively trim every RHOP c-subtree further so that it contains no edge which is a negatively oriented leaf in the RHIP c-tree. (This removes any bt edges by Property 4.16.)
9. Relabel every edge in the final trimmed RHOP c-subtrees as forward.
10. Consider the set of all edges which were cut off from the RHOP c-subtrees in step 6. They form subtrees of the RHOP c-subtrees with *RHOPfirstback* edges as roots. (Remember we are formally dealing with the incidence graph, in which dual edges of G^* are vertices.) Exclusively trim each such subtree so that it contains only *maybeback* edges. (This removes any bt edges defining back twists ending in a forward arc.)
11. For every exclusively trimmed RHOP subtree with *RHOPfirstback* edge as root given by the previous step, find those edges which are also leaves in the RHIP c-tree and find their leaf orientations.

12. Exclusively trim every RHOP subtree from the previous step further so that it contains no negatively oriented RHIP leaf. (This removes any bt edges by Property 4.16.) Label every remaining edge in the trimmed RHOP subtrees as back. Every *RHOPfirstback* edge will also be a back edge.
13. In every RHIP c-subtree, by pointer doubling, find the back edges which have no other back edges as their ancestors in the RHIP c-subtree. Label these edges *RHIPfirstback*. It is possible for a given back edge to be both *RHOPfirstback* and *RHIPfirstback*.

Although we found the forward and back edges using the RHOP c-subtrees, we could also have used the RHIP c-subtrees. Instead of looking at two edges on P', an RHOP path from the root cycle to a leaf, we could have considered two edges on Q', an RHIP path from a leaf to the root cycle. Properties analogous to Properties 4.12, 4.13, 4.14, and 4.15 hold, and we again have the relative ordering of the three types of edges on the c-subtree path from the root cycle to the leaf: if all three types are present, they are arranged forward edges closest to the root cycle, then back edges, then bt edges. A back edge which has no other back edge as ancestor in the RHIP c-tree will be a *RHIPfirstback* edge.

Knowing the sets of forward arcs and back arcs will enable us to find the set of sources with paths to a given non-source vertex in the graph G. We will say that an arc *encloses a source*, if the source has out-edges crossing the root cycle between the left and right endpoints of the RHOP and RHIP paths. If an arc encloses every source in a set of sources, we will say that the arc *encloses the set of sources*. We will also say that an arc *encloses a vertex* if the vertex lies in the region bounded by the RHOP and RHIP paths of the arc and the section of the root cycle from the left endpoint to the right. We will sometimes refer to an arc by the pair of dual vertices (a, b), where a an b are the left and right endpoints of the RHOP and RHIP paths on the root cycle.

The next theorem shows how the forward and back arcs can be used to determine which set of sources have paths to a non-source vertex in G. It forms the basis on which we will construct our procedures to assign vertices to sources in a CS graph.

Theorem 4.9 *Given a non-source vertex x and a source s in a CS graph, x is not reachable from s if and only if there exists a back arc which encloses s, but not x, or a forward arc which encloses x, but not s.*

Proof. (Figure 29) If there exists a back arc enclosing s but not x or a forward arc enclosing x but not s, there can be no directed path from s to x since all graph edges crossing either such arc point away from the region with x in it. The following is the proof that if x is not reachable from s, there exists a forward arc enclosing x but not s or a back arc enclosing s but not x.

Mark all vertices reachable from s. The subgraph G' induced by this set of marked vertices is weakly connected (i.e., connected when viewed as an undirected graph), and all edges between this subgraph G' and the remaining unmarked vertices must point toward G'. Thus the dual of these edges between G' and the unmarked vertices must form a directed path R starting at a dual vertex on the root cycle at some point to the right of the rightmost out-edge of s and ending on the root cycle to the left of the leftmost

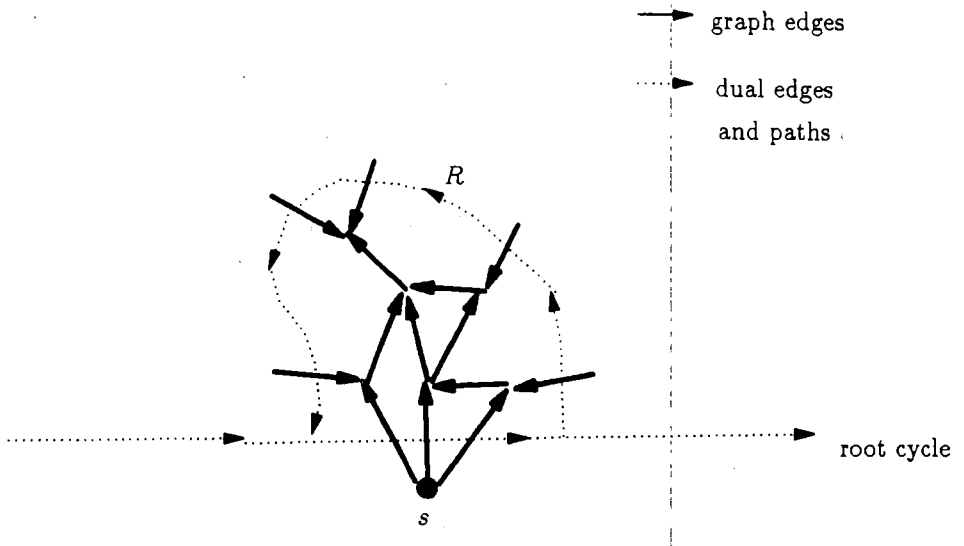


Figure 29
Illustration 1 for Theorem 4.9

out-edge of s. Note that since the unmarked vertices must all be reachable from sources which are not s and we are assuming that all out-edges for a source are in one consecutive group, the marked vertices lie in one hole-free region, and there exists only one such path R. We note that since x cannot be reached from s, it is not enclosed by R.

We now show by contradiction that a dual edge e^* on R cannot be a bt edge. Assume that it is a bt edge and defines a back twist. By definition the sources cannot be inside the twist cycle. By construction, the graph edge e points from a vertex u which cannot be reached from s to a vertex v which can be reached from s. However, by Corollaries 3.4 and 3.5, the RHOP and RHIP paths defining the twist imply two directed paths in G which together surround the twist and meet at u. Hence any path from a source to v inside the twist must imply the existence of a path to u, which contradicts our construction. (Figure 30)

This means that every edge on R defines is either a forward edge or a back edge. We will now show that the regions inside all such forward arcs defined by edges on R and the regions outside all such back arcs defined by edges on R unioned together cover the entire graph G except for the sources and the region enclosed by R. Thus x must lie in one of the regions which is either inside a forward arc or outside a back arc. In either case, an arc separates x from s.

We first note that the RHIP paths and the RHOP paths of the arcs defined by the edges of R may coincide with R, but cannot lie inside the region enclosed by R. This is due to the definition of the RHOP and and RHIP structures; any in-edge or out-edge incident on both a vertex on R and a vertex enclosed by R will lie to the left of the edge on R, and will not be chosen by the RHOP or RHIP path that an edge on R is on. In particular, the endpoints of the RHOP and RHIP paths forming the forward or back arcs are on the part of the root cycle not inside R. (Figure 31)

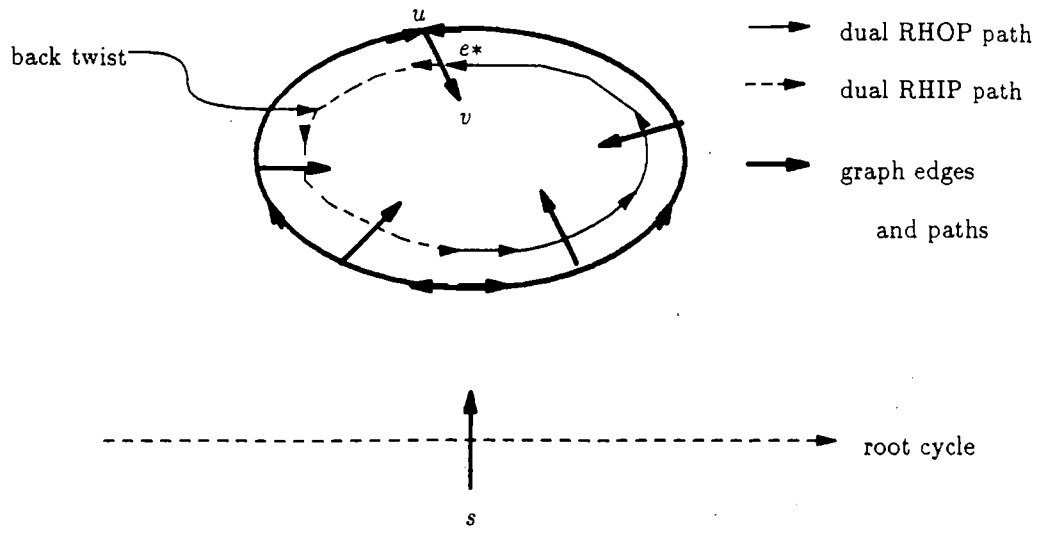


Figure 30
Illustration 2 for Theorem 4.9

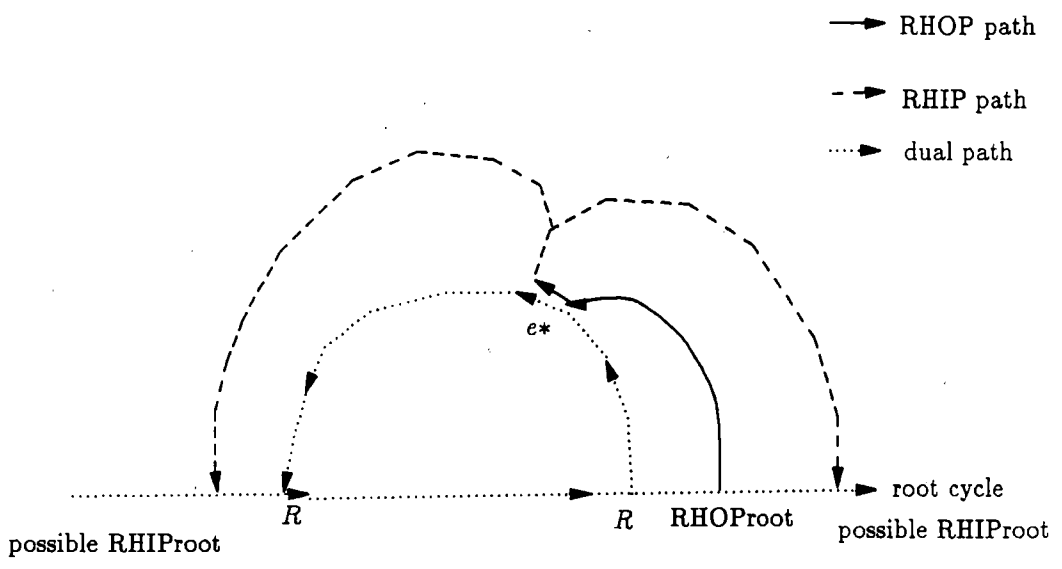


Figure 31
Illustration 3 for Theorem 4.9

Let e_1^* and e_2^* be two adjacent dual edges on R. Let x^* be the dual vertex which is the head of e_1^* and the tail of e_2^*. Let $RHOProot_1$ and $RHIProot_1$ be the endpoints of the RHOP and RHIP paths respectively which form the arc, forward or back, defined by e_1^*. Define $RHOProot_2$ and $RHIProot_2$ similarly for e_2^*. Let *R_1* be the region inside the arc for e_1^* if it is a forward arc and the region outside the arc if it is a back arc. Define *R_2* similarly for e_2^*. (Figure 32)

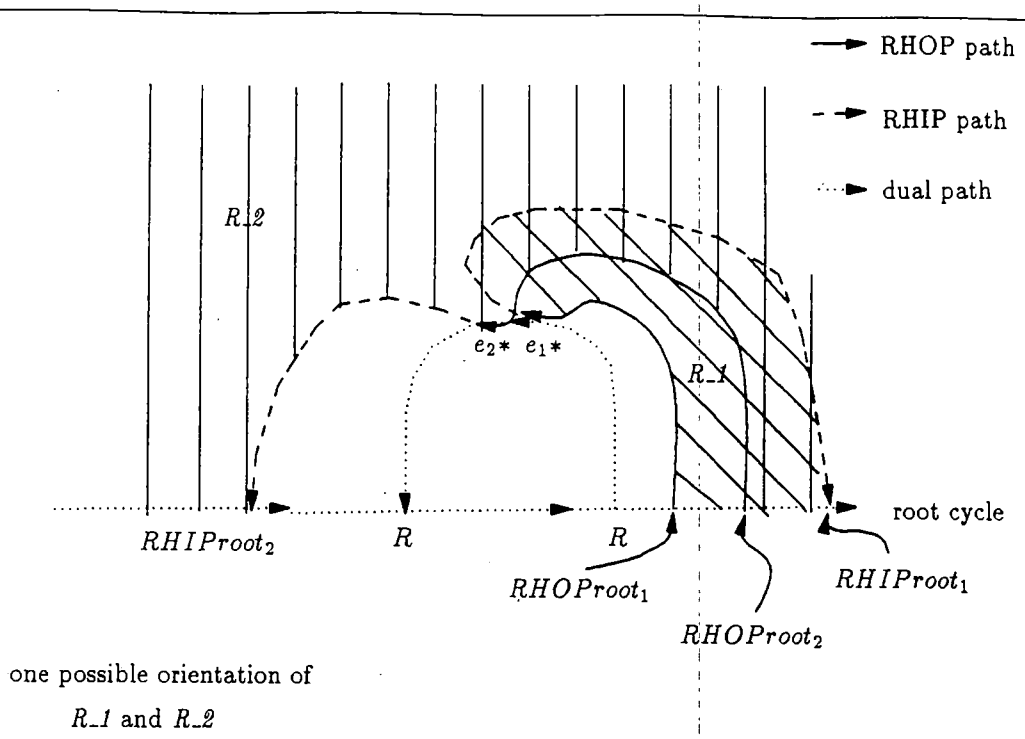


Figure 32
Illustration 4 for Theorem 4.9

We will first show that at least one edge on R is a back edge. Suppose for contradiction that there are no back edges on R. Since all forward arcs must have their RHIP endpoint to the right of their RHOP endpoint, both endpoints must lie either on or to the right of the right endpoint of R or on or to the left of the left endpoint of R. Since the first edge of R containing the starting dual vertex of R must define a forward arc to the right of R, and the the last edge of R must define a forward arc to the left of R (assuming all arcs are forward), there must exist two adjacent edges on R such that one defines a forward arc to the right of R and the other a forward arc to the left of R. Let e_1^* and e_2^* be those two adjacent edges on R. This implies that there is an in-edge f^* to x^* on the RHOP path from $RHOProot_2$ to the left of R. There must also be an out-edge g^* from x^* on the RHIP path to $RHIProot_1$ to the right of R. The relative ordering of edges around x^* must have e_2^*, f^*, g^*, and e_1^* in clockwise order since the RHOP path from $RHOProot_2$ cannot cross the RHIP path to $RHIProot_1$. This implies that x^* is a complex vertex, and we are assuming all dual vertices in the graph are simple. This gives the contradiction, and there must exist at least one back arc on R. (Figure 33)

Since the endpoints of the arc must lie outside the region enclosed by R, we see that if any edge of R defines a back arc, the RHOP endpoint must lie on or to the right of the right endpoint of R on the root cycle, and the RHIP endpoint must lie on or to the left endpoint of R on the root cycle. If we now show that for any two adjacent edges e_1^* and e_2^*, the union of the two regions *R_1* and *R_2* defined as above is a hole-free region, then

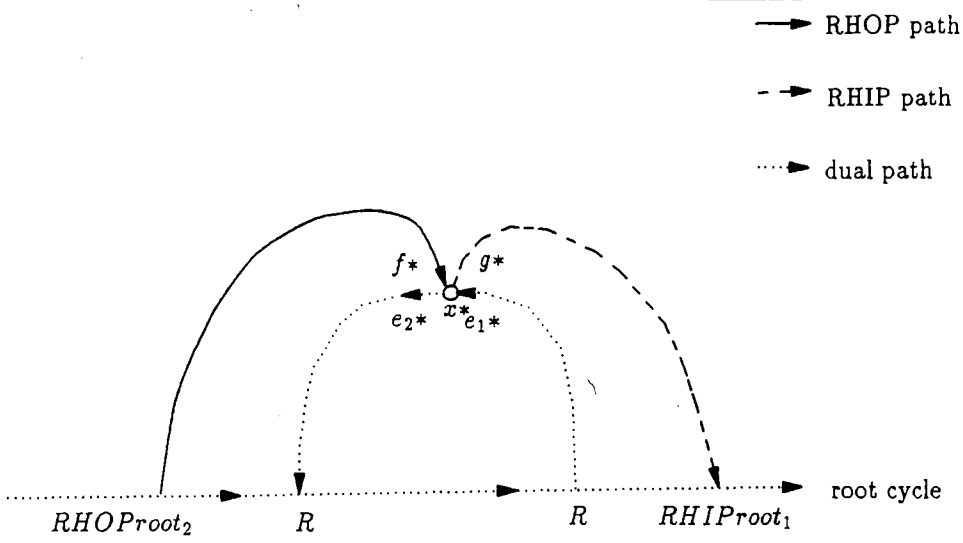


Figure 33
Illustration 5 for Theorem 4.9

the union of all regions inside the forward arcs and outside the back arcs for edges on R must cover the entire graph except the region inside R.

If either the two *RHOProot*s are the same or the two *RHIProot*s are the same for e_1^* and e_2^*, then the two regions must union to a hole-free region. We therefore look at the case where the two edges have different *RHOProot*s and *RHIProot*s. This implies that at x^*, there must be an in-edge f^* on the RHOP path from $RHOProot_2$ and an out-edge g^* on the RHIP path from $RHIProot_1$. As discussed before, the RHOP path from $RHOProot_2$ cannot cross the RHIP path to $RHIProot_1$. Since x^* is a simple vertex, the relative clockwise order of edges around x^* must be e_2^*, g^*, f^*, and e_1^*. This implies that the RHIP path to $RHIProot_1$ must be inside region *R_2*. It cannot cross out because it cannot cross a RHOP path from the right. This means in turn that the two regions *R_1* and *R_2* must form a hole-free region.

Thus the entire region except for the inside of R and the root cycle, must be inside a forward arc or outside a back arc defined by an edge on R. Hence x must be in one of these regions separated from s by that forward or back arc. ∎

We now separate the non-source vertices in a CS graph G into three types. A *type_1* vertex is a vertex in G such that one of its out-edges has as its dual a forward edge. A *type_2* vertex is a vertex which is not type_1 and which has an out-edge whose dual is a back edge. A *type_3* vertex is a vertex in G which is neither type_1 nor type_2. The following two theorems are applications of Theorem 4.9 to the cases of x being a type_1 vertex and a type_2 vertex.

Theorem 4.10 *Given a CS graph, if a type_1 vertex x has an out-edge whose dual counterpart defines a forward arc A, then if there exists a source s which is enclosed by A such that there exists no path from s to x, there exists a back arc B nested inside the forward arc A which encloses s, but not x. A has no edge in common with B.*

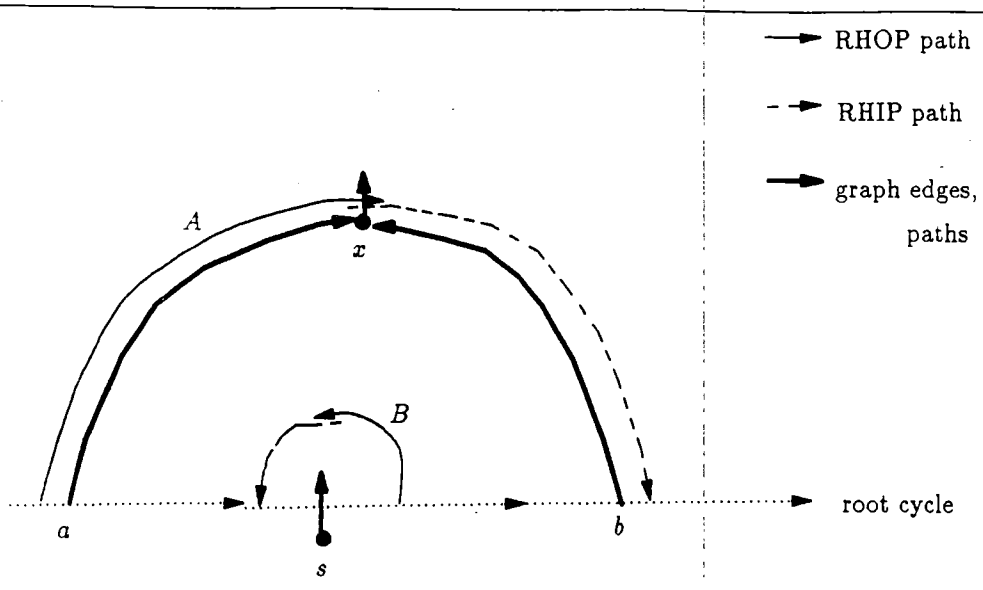


Figure 34
Illustration for Theorem 4.10

Proof. By the previous theorem, we need only show that there exists no forward arc which encloses x, but not s, and that the back arc B enclosing s, but not x, is nested inside A and does not intersect A.

We first show that there exists no forward arc which encloses x, but not s. By Corollaries 3.4 and 3.5, we know there exist directed paths to x from the leftmost source a and the rightmost source b enclosed by the forward arc A. Since any forward arc which encloses x must enclose all sources with a directed path to x, it must enclose a and b. Since s is enclosed by A, it must lie between a and b so that it must also be enclosed by any forward arc enclosing x. Thus no forward arc encloses x, but not s.

We now show that a back arc B which encloses s, but not x, must lie completely inside A. Again let a be the leftmost source and b the rightmost source enclosed by A. Since a and b both have paths to x, and sources inside B cannot have a path to x, a and b must not be enclosed by B. B does, however, enclose a source s which lies between a and b. Thus the endpoints of B must be strictly inside A. Suppose B intersected with A. Since two RHOP paths cannot cross each other, two RHIP paths cannot cross each other, and an RHIP path can cross an RHOP path only from the left to the right, the only way B can cross A is if the RHOP path of B crossed the RHIP path of A and the RHIP path of B crossed the RHOP path of A. This means that x must be inside B, which contradicts our given condition that B encloses s, but not x. Thus A encloses B, and the two arcs do not intersect. (Figure 34) ∎

We can further characterize the back arc B separating x and s by the following corollary.

Corollary 4.11 *If a type_1 vertex x associated with a forward arc A cannot be reached from a source s enclosed by A as in Theorem 4.10, then there exists a back arc B' where*

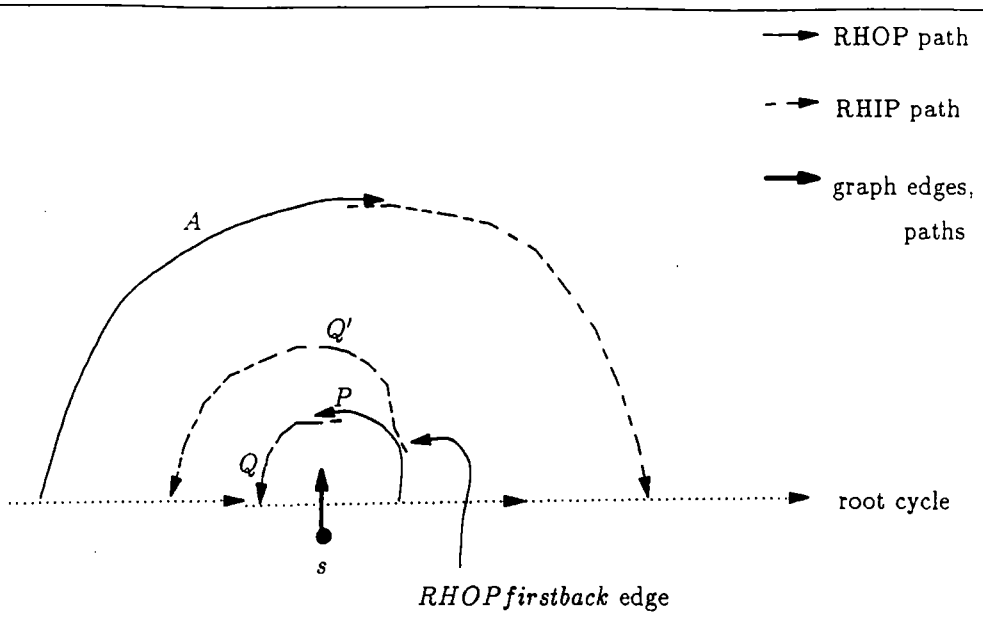


Figure 35
Illustration for Corollary 4.11

B' is associated with a RHOPfirstback edge such that B' encloses s but not x. B' is inside A and does not intersect A. Similarly, x is separated from s by a back arc B'' where B'' is associated with a RHIPfirstback edge.

Proof. (Figure 35) From Theorem 4.10 we know that there exists a back arc B which encloses s, but not x, and B is nested inside A. If B is associated with a *RHOPfirstback* edge, we are done proving the first part of the corollary. If not, let e^* be the back edge defining B, and let P and Q be the RHOP and RHIP paths forming B. Then there must be a *RHOPfirstback* edge on P between the root cycle and e^*, and by Property 4.13, the associated back arc B' must have its RHIP path Q' end on the root cycle to the to the left of the endpoint of Q. Q' cannot cross A, because one RHIP path cannot cross another and a RHIP path cannot cross a RHOP path from the right. Hence B' is inside A and does not intersect it. It encloses B and hence s, but not x. We can use similar arguments for the back arc B'' associated with the *RHIPfirstback* edge on Q. ∎

Theorem 4.12 *In a CS graph, let v_{top} be a type_1 vertex associated with a forward arc A. Let x be a type_2 vertex enclosed by A such that there exists no forward arc which encloses x but not v_{top}. Let B be the back arc associated with x. If there exists a source s not enclosed by B but enclosed by A which has no path to x, then s is enclosed by some back arc B' which is enclosed by A and intersects neither A nor B.*

Proof. (Figure 36) Let a and b be respectively the leftmost source and the rightmost source enclosed by A. Let c be the source whose out-edge crosses the root cycle immediately to the left of the RHIP endpoint of B. Let d be the source whose out-edge crosses the root cycle immediately to the right of the RHOP endpoint of B. By

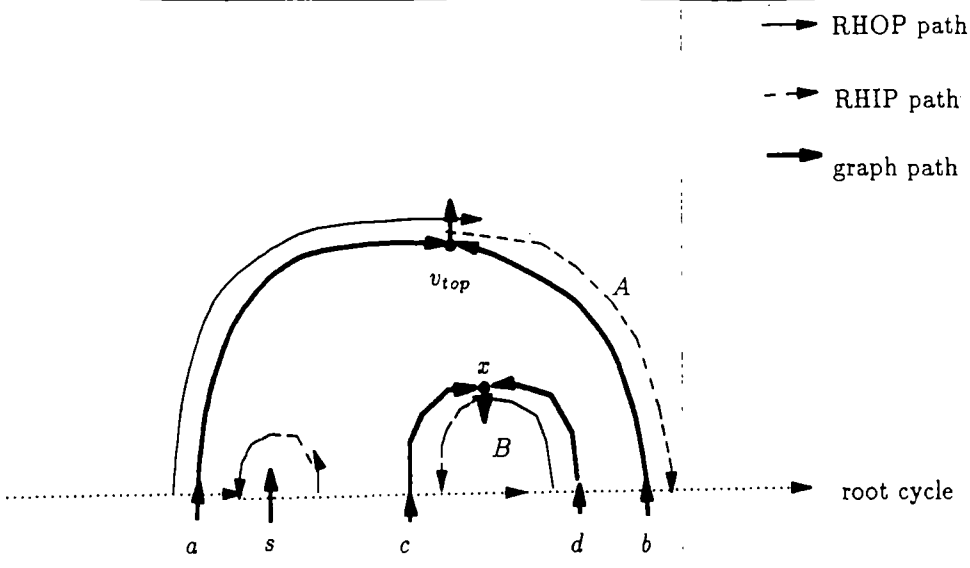


Figure 36
Illustration for Theorem 4.12

Corollaries 3.4 and 3.5, there exist paths to v_{top} from a and b and paths to x from c and d. Since s is enclosed by A but not by B, there are two possible cases: s can lie to the left of c or to the right of d. Since s has no path to x, it cannot be either c or d.

Let the case when s lies to the left of c be case 1. By Theorem 4.9, there must exist a forward arc A' which encloses x but not s or a back arc B' which encloses s but not x. We will first show by contradiction that no such A' exists.

Suppose A' exists. Since it encloses x, it must enclose all sources with paths to x so that it must enclose c and d. Since A' does not enclose s, its left RHOP endpoint must lie on the root cycle between the edges crossed by the out-edges of s and c. Thus it cannot enclose a. A' then cannot enclose v_{top}, since if it did, it must also enclose a. Then A' is a forward arc which encloses x but not v_{top}, but one of the conditions of the theorem is that such a forward arc does not exist. Thus A' does not exist.

Thus there exists a back arc B' which encloses s but not x. First we show that B' cannot intersect B. Since B' cannot enclose c, its endpoints must be separate from those of B and lie to their left. The only way the two back arcs can then intersect would be if the RHOP path of B' crossed the RHIP path of B from the right, and if the RHIP path of B' crossed the RHOP path of B from the left. However, this would mean that x is inside B', so that the two back arcs cannot intersect.

Suppose B' intersected A. Since the right RHOP endpoint of B' must be inside A, the only way B' could intersect A is if the RHOP path of B' crossed the RHIP path of A from the right. However, this would create a forward arc separating v_{top} and x, which cannot exist. Hence B' does not intersect A, but lies inside A. We note that the left endpoint of B' and the let endpoint of A cannot coincide since this would create a complex vertex at the endpoints, and in a CS graph all dual vertices are assumed simple.

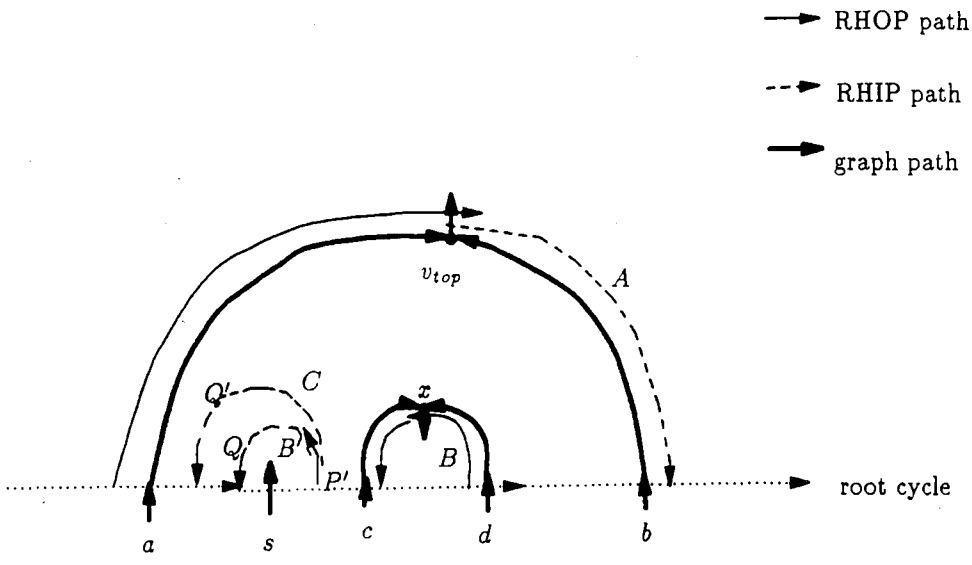


Figure 37
Illustration for Corollary 4.13

When s lies to the right of d, we will call it case 2. The same argument can be made as in case 1 above, by switching (left) and (right), (RHOP) and (RHIP), (c)and (d), and (a) and (b). ∎

Corollary 4.13 *Suppose we had the same conditions as in Theorem 4.12 case 1, where s lies to the left of B. Then there exists a back arc C associated with a RHOPfirstback edge which encloses s but not x. Again C is inside A but outside B, and intersects neither A nor B.*

Proof. (Figure 37) By Theorem 4.12, there exists a back arc B' which encloses s but not x, and is inside A and outside B. Let P and Q be respectively the RHOP and RHIP paths of B'. There must exist a *RHOPfirstback* edge e^* on P. Let C be the back arc associated with e^*, and let P' and Q' be the RHOP and RHIP paths of C. By Property 4.13, the endpoint of Q' lies to the left of the endpoint of Q. Thus B' is inside C, and C encloses s.

Q' cannot intersect B because two RHIP paths cannot cross and an RHIP path cannot cross an RHOP path from the right. Thus x cannot be enclosed in C. C cannot intersect A for the same reasons given in the proof of Theorem 4.12 that B' cannot intersect A. ∎

Corollary 4.14 *Suppose we had the same conditions as in Theorem 4.12 case 2, where s lies to the right of B. Then there exists a back arc D associated with a RHIPfirstback edge which encloses s but not x. Again D is inside A but outside B, and intersects neither A nor B.*

Proof Sketch. (Figure 38) Again the proof is the same as for the previous Corollary 4.13, with the following switches: (left) and (right), (RHOP) and (RHIP), (P) and (Q), and (C) and (D). ∎

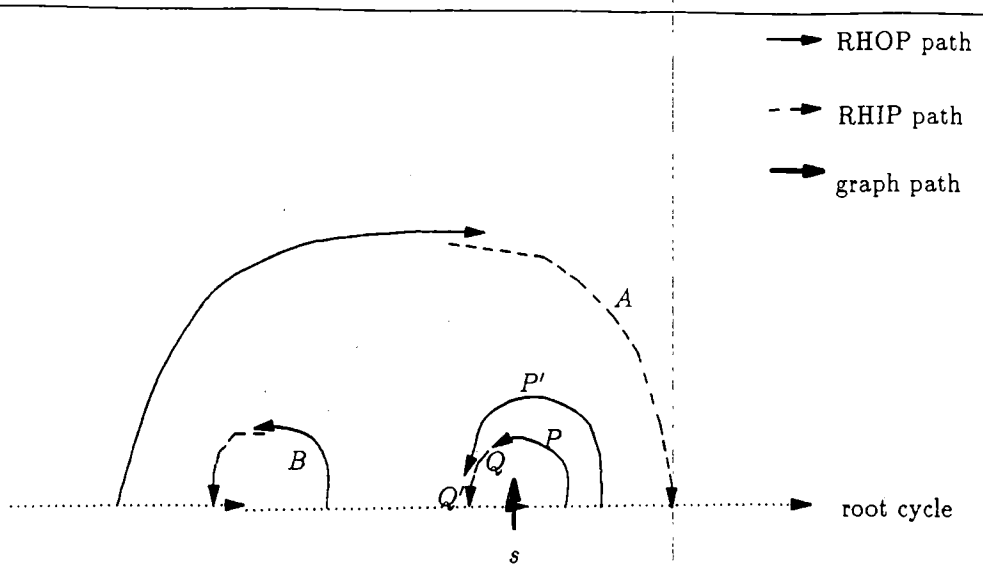


Figure 38
Illustration for Corollary 4.14

4.4.3. Finding the hole structure

Because of Theorem 4.9, if we find for any vertex x the set S of largest back arcs which do not enclose x and the smallest forward arc A which does enclose x, we know that all sources outside the back arcs of S and inside A have paths to x. This section shows how the back arcs can be organized so that we can quickly find, for a given vertex, which back arcs enclose it and which do not, and which source out of a set of sources enclosed by a back arc has the highest priority. For this we find the hole structure of the graph.

We wish to partition the graph into subgraphs such that all vertices in a given subgraph are enclosed by one back arc B, but not enclosed by any back arc which is inside B. This will give a nesting structure to the partitions. A subgraph defined in this way will be called a *hole* defined by the arc B. A back arc B_1 will be *inside* a back arc B_2 if all vertices enclosed by B_1 are also enclosed by B_2. A vertex v *belongs* to a hole H if v is a vertex in the subgraph H. We will say a *source s belongs to hole H* if s has an out-edge to a vertex belonging to H. We note that a given source may belong to more than one hole by this definition, but the number of holes it belongs to will never exceed the number of out-edges for the source.

There is a potential problem with trying to partition the graph this way. Two back arcs can cross so as to create an intersection region, and neither arc can be said to be inside the other. Fortunately, because of Corollaries 4.11, 4.13, and 4.14 we do not need consider all possible back arcs. We will make two separate hole structures: one using the set of back arcs associated with *RHOPfirstback* edges (abbreviated *RHOPhole structure*) and one using the set of back arcs associated with *RHIPfirstback* edges (abbreviated *RHIPhole structure*). The following lemma shows that in each of the two hole structures, the back arcs do not cross, so that we can define a nesting relationship among the holes.

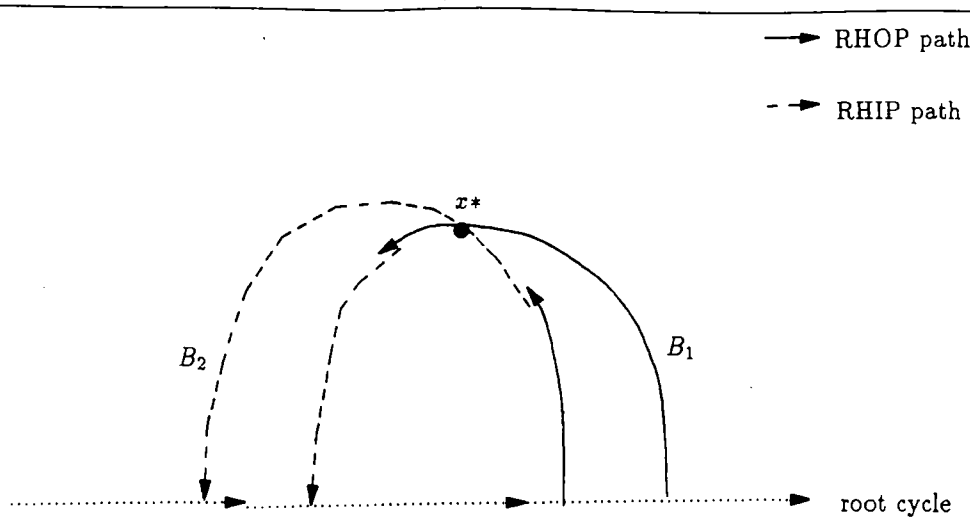


Figure 39
Illustration for Lemma 4.15

Lemma 4.15 *Let B_1 and B_2 be two back arcs associated with RHOPfirstback edges. Then the path of B_1 and the path of B_2 never cross. Similarly, if B_1 and B_2 are both back arcs associated with RHIPfirstback edges, they do not cross each other.*

Proof. (Figure 39) When two back arcs cross, they must cross where the RHOP path of one is crossed from the left by the RHIP path of the other. Without loss of generality, let the RHOP path of B_1 be crossed from the left by the RHIP path of B_2 at vertex x^*. Then there exists a back arc B formed by the RHOP path of B_1 to x^* and the RHIP path of B_2 from x^*. Thus B_1 cannot have been defined by a *RHOPfirstback* edge, and B_2 cannot have been defined by a *RHIPfirstback* edge. ∎

We can further simplify the set of back arcs in each structure by considering only the smallest back arc from a set of back arcs which have the same two endpoints. The following lemma follows directly from the definitions of the terms used.

Lemma 4.16 *Let back arc B_1 and back arc B_2 have the same endpoints, and B_2 be inside B_1. Suppose there exist a vertex x and a source s such that B_1 encloses s, but not x. Then B_2 encloses s, but not x.*

We will need to determine the nesting structure for the holes quickly. Because of the following lemma, we know that if a dual edge e^* defines a back arc which we use to define a hole G, so that the head of the graph edge e belongs to hole G and the tail of e belongs to a different hole H, then H is the parent of G in the nesting order. The lemma eliminates the possibility that there exists another hole I which is the proper ancestor of G and a proper descendant of H in the nesting order.

Lemma 4.17 *Let B_1 be a back arc defined by a dual edge e^*. Let x be the vertex which is the tail of the graph edge e crossing e^*. If B_1 is inside a back arc B_2, then x is enclosed by B_2.*

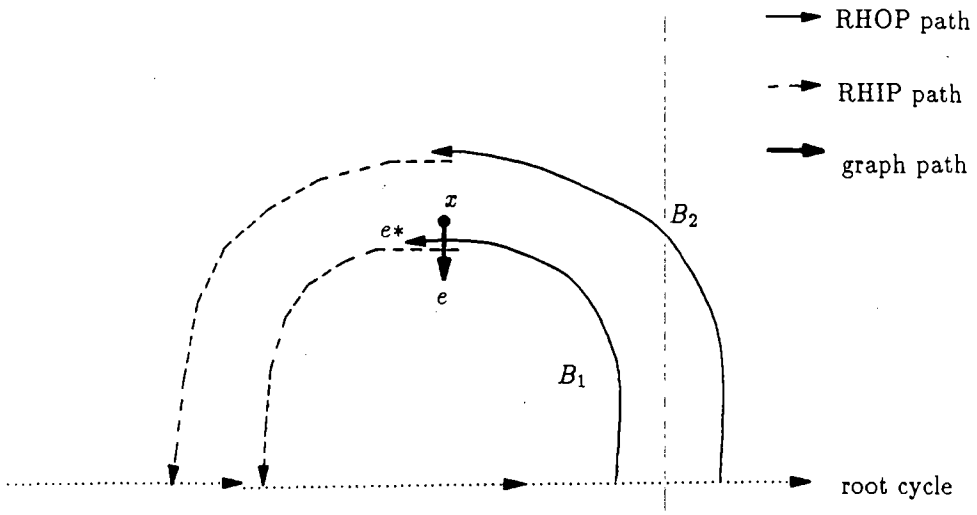


Figure 40
Illustration for Lemma 4.17

Proof. (Figure 40) We will give a proof by contradiction. Suppose B_2 does not enclose x. Since B_1 is inside B_2, B_2 must enclose the head of e. Thus e^* must be an edge on the back arc B_2. Then either the RHOP path of B_2 is a continuation of the RHOP path of B_1, or the RHIP path of B_2 is a backward continuation of the RHIP path of B_1. However, we know in both cases the continuations must be inside B_1 by Property 4.13 and an equivalent property for RHIP paths. Hence B_2 must be inside B_1, giving a contradiction. Hence, x must be enclosed by B_2. ∎

Let G' be the subgraph of a CS graph G induced by all vertices which are not enclosed by any of the back arcs in a hole structure. We will define this subgraph to be a hole, assuring that there will be a single root hole in the nesting tree.

By using all the above we get procedures for finding holes and creating a nesting tree for them. We give the procedure for finding the RHOPhole structure of a CS graph G. Finding the RHIPhole structure for G is done by switching the roles of the RHOP and RHIP c-trees in the procedure.

procedure *FindRHOPHoles*
(* Given a CS graph G with dual edges labelled by procedure *LabelForwardBack*, find and identify the holes in the RHOPhole structure, and create a nesting tree for them.*)
1. Inclusive trim each RHOP c-subtree so that all edges labelled *RHOPfirstback* by *LabelForwardBack* are leaves in the trimmed RHOP c-subtrees and they are the only leaves for the trimmed RHOP c-subtrees.
2. Inclusive trim each RHIP c-subtree, discarding any branch ends which do not contain any edges labelled *RHOPfirstback* by *LabelForwardBack*.
3. On every trimmed RHOP c-subtree, partition all RHOP c-subtree leaves according to the RHIP c-subtrees to which they also belong. Each partition set contains back

edges whose arc endpoints are all the same. Remove the *RHOPfirstback* label from all such RHOP c-subtree leaves except the one defining the smallest back arc (the last such RHOP c-subtree leaf in each partition set in a right hand Euler tour of a RHOP c-subtree).

4. Inclusive trim both RHOP and RHIP c-subtrees again, discarding any branch ends which no longer have contain *RHOPfirstback* edges as leaves.
5. In G, cut all edges crossing the edges of either a trimmed RHOP c-subtree or a trimmed RHIP c-subtree. Cut all edges crossing the root cycle.
6. Run the algorithm for finding connected components on the undirected version of the modified G. For every *RHOPfirstback* edge e^*, label the connected component which includes the head of the edge e crossing the dual e^* as a hole defined by e^*.
7. Label all non-source vertices of G which are not part of holes found in the previous step as vertices belonging to the root hole.
8. For every hole, identify its parent in the nesting tree by finding to which hole the tail of the edge e crossing the *RHOPfirstback* edge e^* belongs.
9. For every hole, find the sources belonging to it by identifying the sources which have an out-edge crossing the root cycle and incident on the connected component.

All steps in the procedure can be done in $O(\log n)$ time with $O(n)$ processors in the CRCW PRAM model where n is the number of vertices in the graph.

Note that because no two back arcs have the same endpoints, every hole must have at least one source belonging to it. The importance of the nesting tree is that if a vertex belongs to a hole, all sources belonging to holes which are not its ancestors cannot have paths to the vertex, by Theorem 4.9. If a source belongs to an ancestor hole, it may or may not have a path to the vertex, depending on whether or not there exists a forward arc enclosing the vertex but not the source.

4.4.4. Assigning sources for type_1 vertices

In this subsection we will show how to create a data structure on the hole structure and use it to assign sources for the type_1 vertices. By Corollary 4.11, we can use either the RHOPhole structure or the RHIPhole structure. We will choose for concreteness the RHOPhole structure; all lemmas and properties which we give below for the RHOPhole structure will have its counterpart in the RHIPhole structure. Suppose we look at a type_1 vertex x which belongs to a hole H in the RHOPhole structure. Let A be the forward arc associated with x, and let (a, b) be the pair of vertices on the root cycle, where a is the left RHOP endpoint of A and b is the right RHIP endpoint of A. Then by Corollary 4.11, the sources belonging to ancestor holes of H in the nesting tree (including non-proper ancestors, i.e., H) which have out-edges crossing the root cycle between a and b, will have paths to x.

Let every back arc defining a hole in the RHOPhole structure be identified by the pair of its endpoints (c, d). Note that this identification is unique since we kept only the smallest back arc out of all those with the same endpoints. Then the following lemma will let us identify the highest ancestor in the nesting tree with sources which have paths to x.

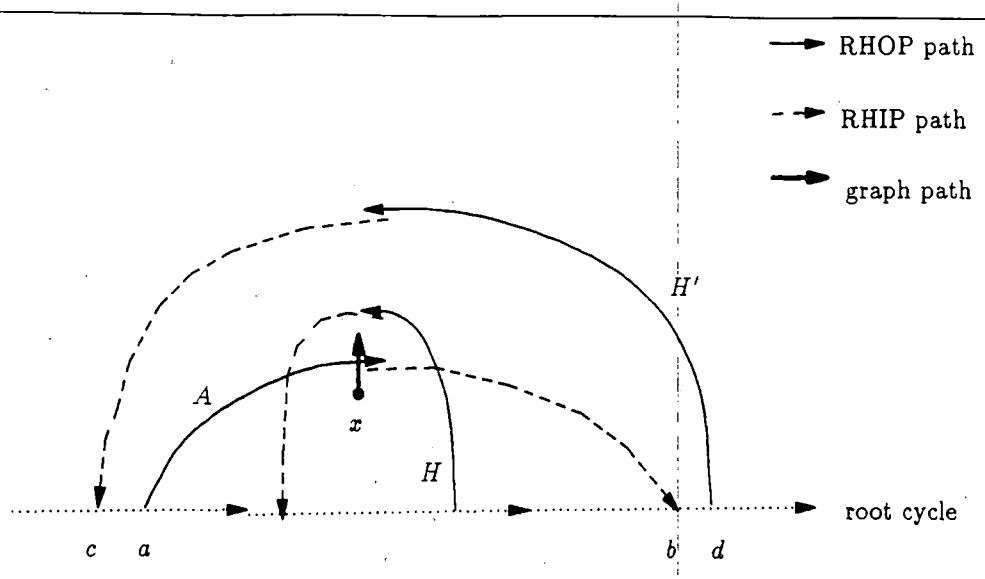


Figure 41
Illustration of Lemma 4.18

Lemma 4.18 *In a CS graph, let x be a type_1 vertex in hole H with a forward arc A with endpoints (a, b) as in the discussion above. Let the back arc (c, d) in the RHOPhole structure be the smallest back arc in the hole structure in which c is to the left of a and d is to the right of b. (More formally, (c, d) has this property and is inside all other back arcs with this property.) Then the hole H' defined by (c, d) is an ancestor of H and there exist no paths to x from a source belonging only to proper ancestors of H' in the nesting tree.*

Proof. (Figure 41) Suppose H' were not an ancestor of H. Then there can be no paths to x from sources enclosed by (c, d). However, we know that there exists at least one source s enclosed by A with a path to x, and since c is to the left of a and d is to the right of b, (c, d) must also enclose s, giving a contradiction.

Suppose source s belonged only to proper ancestors of H'. Then the forward arc A would enclose x by definition of a type_1 vertex, but not s since the arc endpoints of A lie inside H'. Thus there can be no paths to x from s. ∎

The following property which characterizes the intersection of a forward arc and a back arc will allow us to find the hole H' of Lemma 4.18 for a given forward arc.

Property 4.17 *Let us say that the left RHOP endpoint a of a forward arc A lies in hole I if the back arc (c, d) which defines I is the smallest back arc in the RHOPhole structure for which $a = c$ or a lies between c and d. Similarly the right RHIP endpoint b of A lies in J if the back arc (c', d') which defines J is the smallest back arc in the RHOPhole structure for which $b = d'$ or b lies between c' and d'. Then the type_1 vertex x which defines A will be enclosed by both (c, d) and (c', d').*

Proof Sketch. (Figure 42) This follows since a RHOP path cannot cross out of (c, d); nor can a RHIP path cross into (c', d'). ∎

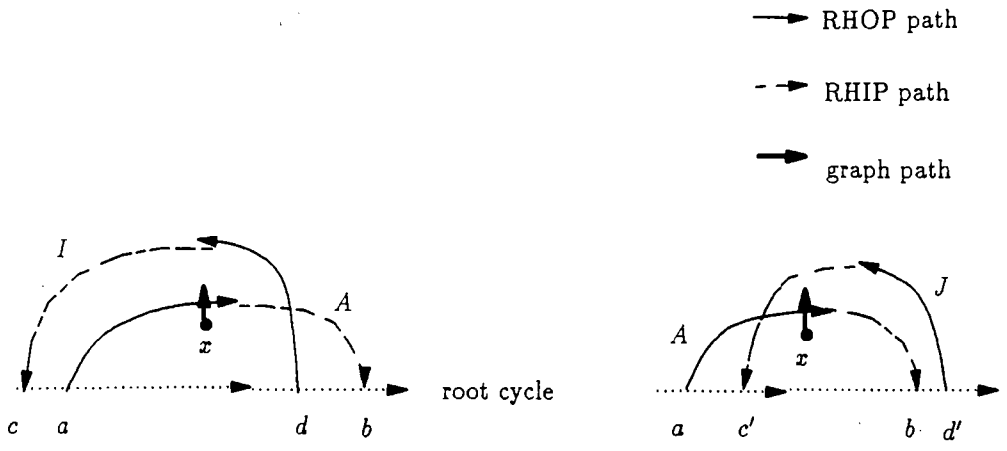


Figure 42
Illustration of Property 4.17

Since both (c, d) and (c', d') enclose x, I and J will be the same or in a descendant-ancestor relationship to each other. The one which is the ancestor will therefore be the hole H' in Lemma 4.18. We will call I the $LeftAncestorHole$ of x, J the $RightAncestorHole$ of x, and H' the $HighestAncestorHole$ of x.

Let a type_1 vertex x belong to hole H, and let the holes in the nesting tree path from $HighestAncestorHole$ to H be those defined by the series of back arcs $(c_k, d_k), (c_{k-1}, d_{k-1}), \ldots (c_1, d_1)$. For concreteness, let us assume that $RightAncestorHole$ is the same as $HighestAncestorHole$, and is defined by (c_k, d_k), $LeftAncestorHole$ is defined by (c_j, d_j) where $1 < j \leq k$, and H is defined by (c_1, d_1). We can then divide the sources with paths to x into four sets. (Figure 43)

The first set A is composed of the sources belonging to H and hence lying between c_1 and d_1. (We will say a source *lies between a and b* if the source has an out-edge crossing the root cycle between vertices a and b on the root cycle.) The second set B consists of sources lying between d_i and d_{i+1} for $i = 1$ to $k - 2$. The third set C consists of sources lying between c_i and c_{i+1} for $i = 1$ to $j - 2$. The last set D has the sources in $RightAncestorHole$ lying between d_{k-1} and d_k and in $LeftAncestorHole$ lying between c_{j-1} and c_j.

We note that all sources in a given hole in set B lie in a range whose right boundary is the right boundary of the hole. Similarly, all sources in a given hole in set C lie in a range whose left boundary is the left boundary of the hole. If we preprocess using the prefix maximum algorithm to find the highest priority source lying in a range whose boundary is either the left or right boundary of a hole, one processor can find in constant time the highest priority source lying between c_i and c_{i+1} or between d_i and d_{i+1}. We will call this the $BestLeftSource$ or $BestRightSource$ associated with the edge in the hole nesting tree connecting the hole defined by (c_i, d_i) and (c_{i+1}, d_{i+1}). Further preprocessing as will enable us to determine the highest priority source among all the $BestLeftSources$ in C and the highest priority source among the $BestRightSources$ in B. We will thus be

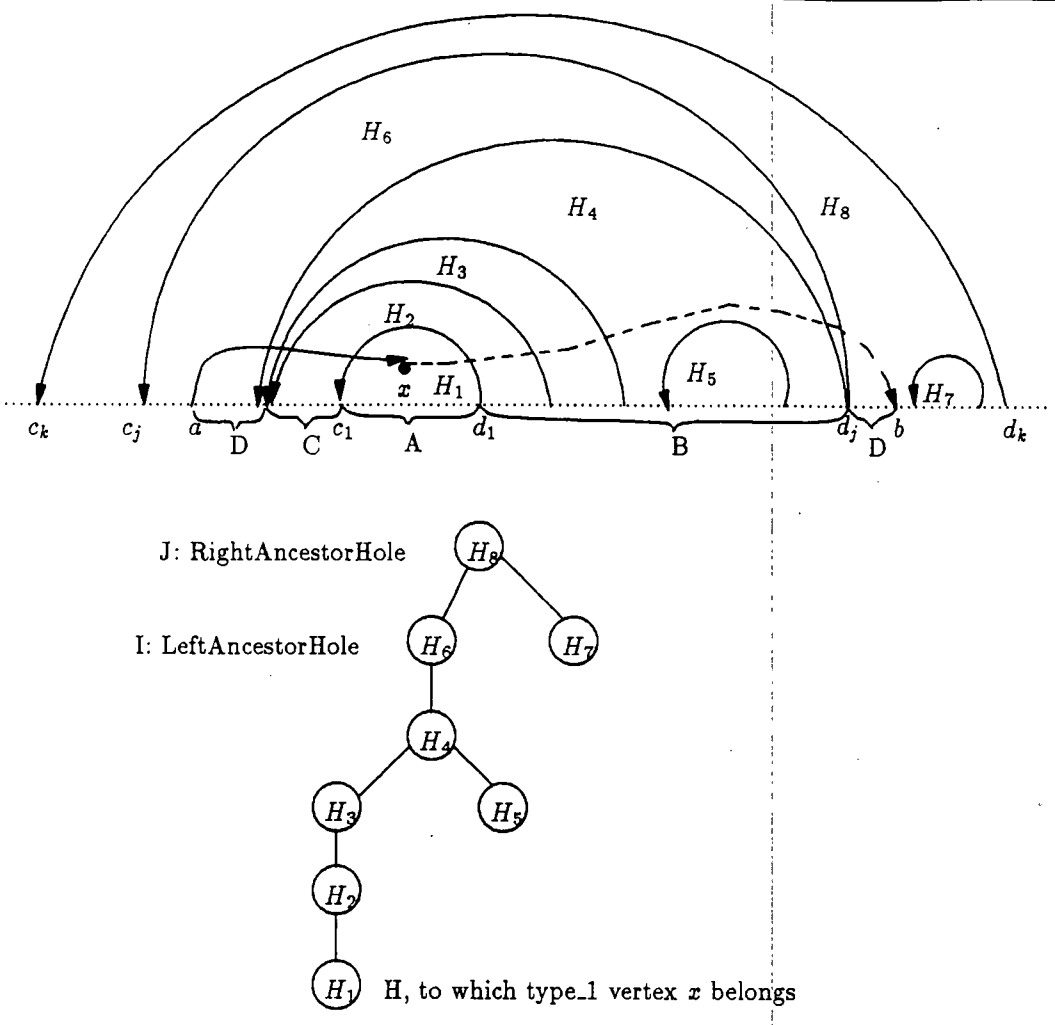


Figure 43
Sources with paths to x

able to determine the highest priority source in each of the four sets of sources in $O(\log n)$ time using a single processor.

One minor subcase occurs when one of the two ancestor holes, say the *RightAncestorHole*, is the same as hole H that vertex x is in. In this case we will say that both sets A and B are empty, and include all sources in H with paths to x in set D. If both endpoints of the forward arc lie in the same hole H that x belongs to, all sets except D are empty.

4.4.5. Range tree data structures for hole structures

The previous subsection gives the basic plan used to assign to a type_1 vertex x the highest priority source s with a path to x. Starting with the nesting tree of holes, we build a data structure around it so that we can find the highest priority source among the *BestLeftSources* and the *BestRightSources* associated with a path segment of the tree. In order to build this data structure, which we will call *RHOPhole tree*, we need to use

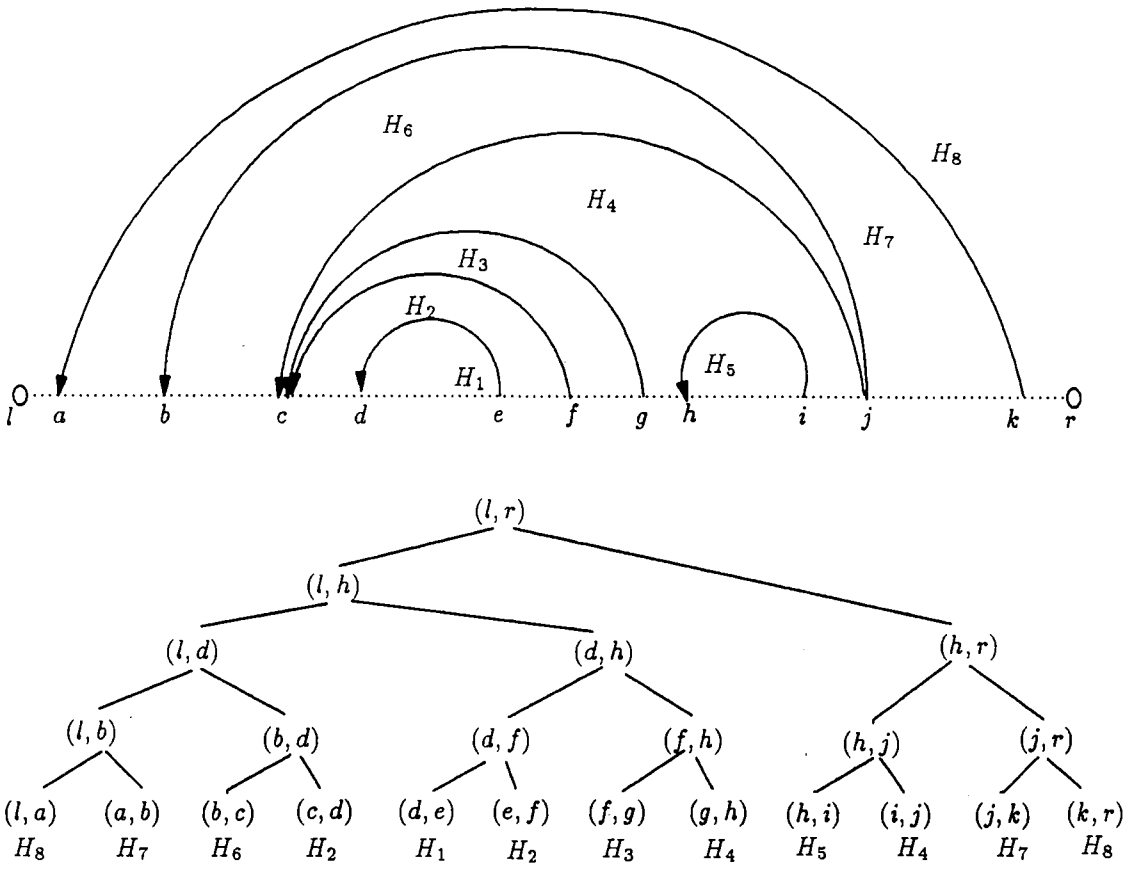


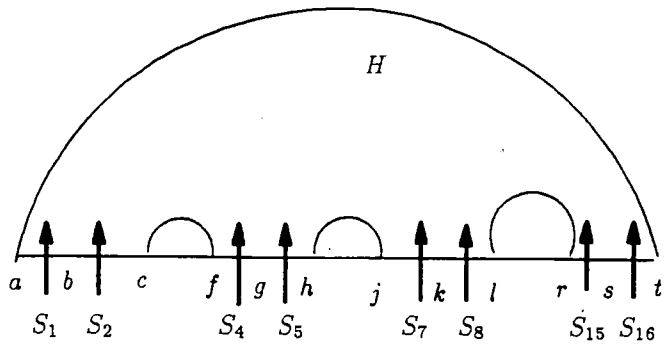
Figure 44
Range tree for holes

two range trees, one for holes and the other for sources within a hole. These range trees will also enable us to find the highest priority sources for sets A and B. They will be built using the following two procedures.

procedure *BuildRangeTreeForHoles*

(* This procedure takes the set of back arcs and holes found by the procedure *FindRHOPHoles* and builds a range tree such that given any endpoint a of an arc on the root cycle, we can find which hole it lies in. *) (Figure 44)

1. Take all endpoints of the back arcs and sort them in left to right order. Add a special vertex to be the leftmost vertex and another to be the rightmost. Define each pair of adjacent vertices in the sorted list to be a *basic interval.*
2. For each basic interval, identify the hole it belongs to; i.e., if a source has an out-edge which crosses the root cycle between vertices a and b where a and b define a basic interval, then the hole to which the head of the out-edge belongs is the hole to which the basic interval (a, b) belongs.
3. Build an ordered essentially complete binary tree with the basic intervals in sorted order as the leaves. (See [BB88, p. 25] for the definition of essentially complete



Priority ordering

$S_8 > S_5 > S_1 > S_{15} > S_2 > S_{16} > S_4 > S_7$

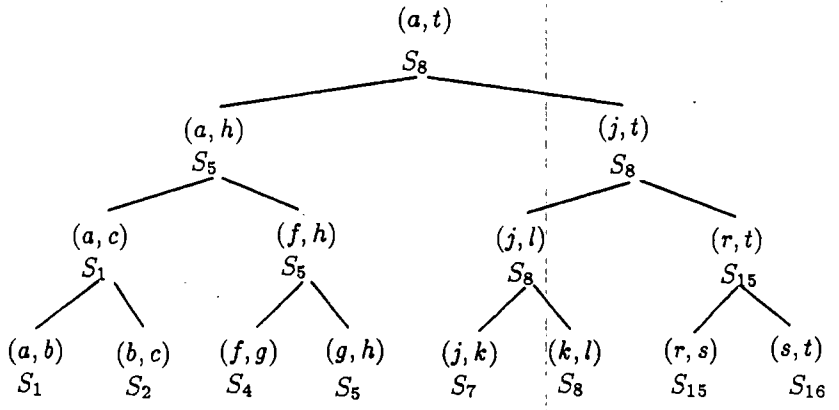


Figure 45
Range tree for sources

binary tree.) Each interior node in the tree defines an interval which is the union of the two contiguous intervals which are its children.

procedure *BuildRangeTreeForSources*

(* This procedure takes a hole found by procedure *FindRHOPHoles* and builds a range tree such that, given any pair of vertices on the root cycle, we can find the highest priority source, if any, which belongs to the hole and which lies between the two vertices given.*) (Figure 45)

1. Mark the root cycle edges crossed by an out-edge from a source to a vertex belonging to the hole.
2. Let the endpoints of the marked edges from step 1 define the leaf intervals for the source range tree. (Each leaf interval is one edge on the root cycle.) Associate with every leaf node the source whose out-edge crossed the defining root cycle edge. Order the leaf intervals in the same left to right order as on the root cycle.

3. Build an ordered essentially complete binary tree on the leaves, such that the parent node defines the union of the two intervals of the children and the associated source is the higher priority source of the two sources associated with the children. If the two intervals of the children are not contiguous, use the interval with the left endpoint of the left child's interval and the right endpoint of the right child's interval.

All steps for both procedures can be done in $O(\log m)$ time using $O(m)$ processors where m is the number of edges in the subgraph involved. Note that the total number of holes is never more than the number of out-edges of type_1 vertices.

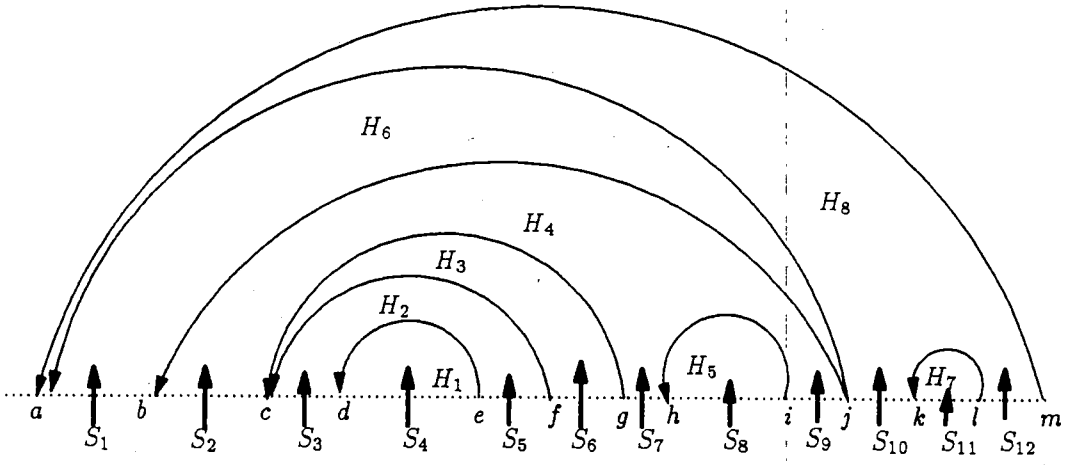
The range trees are used in the standard way to answer queries of the type, "Given a forward arc with endpoints (a, b), in what holes do the endpoints lie?" and "Given an interval (a, b), what is the highest priority source belonging to a given hole which lies between a and b?". All such queries can be answered by a single processor in $O(\log m)$ time where m is the number of edges in the subgraph involved.

We now give the procedure for building the RHOPhole tree.

procedure *BuildRHOPHoleTree*

(* Given the holes and their nesting trees resulting from procedure *FindRHOPHoles*, and the range trees for the holes and for sources for every hole, we build a RHOPhole tree.*) (Figure 46)

1. For every hole, using the range tree for sources, build two tables which contain the results for the prefix maximum problems defined as follows. The value $A[k]$ in the prefix maximum problem will be the source associated with the interval k where the intervals are those which were the leaves in the range tree for sources. In the *left-range table*, the intervals are considered ordered from left to right, and in the *right-range table*, the intervals are ordered right to left. The value S_i stored in the left-range table is the highest priority source for all intervals from the leftmost to the i^{th} from the left. Similarly for the right-range-table, the value S_i is the highest priority source for all intervals from the rightmost to the i^{th} from the right. The highest priority source belonging to the hole can be found from either table.
2. For every edge in the nesting tree connecting a child hole (i, j) with a parent hole (k, l), find two sources, a ***BestLeftSource*** and a ***BestRightSource***. The ***BestLeftSource*** is the highest priority source belonging to the parent hole (k, l) which lies in the interval (k, i). The ***BestRightSource*** is the highest priority source belonging to the parent hole (k, l) which lies in the interval (j, l). Both can be found from the tables created in step 1.
3. Find for each hole in the nesting tree, the distance of the hole from the root of the nesting tree. Record this as the $level(H)$ of the hole H.
4. By pointer doubling, find for every hole H in the tree, a pointer to the hole which is 2^j levels above it in the path to the root of the tree for all j from 0 to $\lfloor level(H) \rfloor$. Find at the same time for each path from hole H to the hole 2^j levels above it, the highest priority source encountered among the ***BestLeftSources*** for the edges along the path. Also find the highest priority source among the ***BestRightSources*** encountered along each path. We thus create a table for each hole H in the tree.



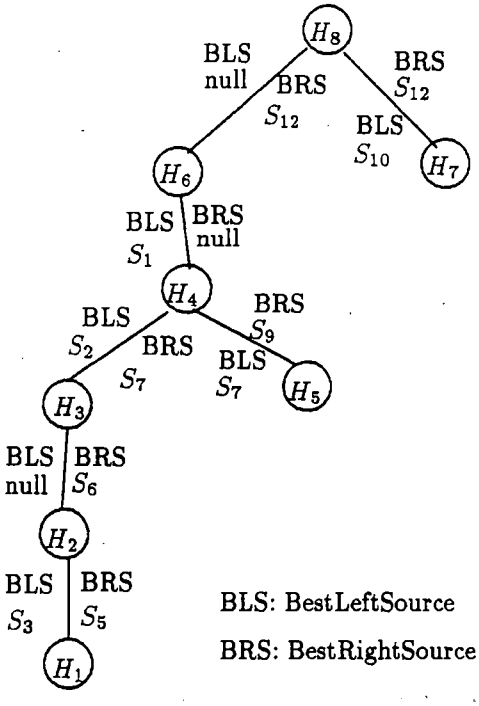
Priority ordering: $S_6 > S_7 > S_1 > S_4 > S_3 > S_{11} > S_8 > S_{12} > S_2 > S_{10} > S_9 > S_5$

Left-range table for H_4

	(b,c)	(b,h)	(b,j)
highest priority source	S_2	S_7	S_7

Right-range table for H_4

	(i,j)	(g,j)	(b,j)
highest priority source	S_9	S_7	S_7



BLS: BestLeftSource

BRS: BestRightSource

RHOPhole tree

Table for H_1

path length	hole	BLS	BRS
2^0	H_2	S_3	S_5
2^1	H_3	S_3	S_6
2^2	H_6	S_1	S_6

Highest priority source in H_1: S_4

Figure 46
RHOPhole tree

The table has $\lfloor level(H) \rfloor$ entries, and each entry consists of a pointer, and two sources. The RHOPhole tree consists of the nesting tree supplemented by these tables created for each hole in the tree. For each hole node in the tree, also record the highest priority source belonging to that hole.

Every step in the above procedure can be done in $O(\log m)$ time using $O(m)$ processors where m is the number of edges in the graph. Note that the nesting tree has as its nodes the holes defined by back arcs in an RHOPhole structure. The total number of all intervals in all tables for holes in step 1 is the number of edges on the root cycle.

We are now ready to assign a source to each type_1 vertex.

procedure *AssignType1Vertex*

(* This procedure takes a CS graph G and assigns for each type_1vertex x in G the highest priority source with a path to x.*)

1. Run procedure *LabelForwardBack*.
2. Run procedure *FindRHOPHoles*. Let h be the height of the nesting tree.
3. Run procedure *BuildRangeTreeForHoles*.
4. Run procedure *BuildRangeTreeForSources* for each hole.
5. Run procedure *BuildRHOPHoleTree*.
6. Mark every edge in G which is crossed by a dual edge marked *forward* in procedure *LabelForwardBack*.
7. For every vertex u with one or more marked out-edges, choose one such out-edge and its associated forward arc (a_u, b_u). Vertex u is thus a type_1 vertex.
8. For every type_1 vertex u, find the hole $hole(u)$ to which it belongs from *FindRHOPHoles*.
9. Using the hole range tree, for every type_1 vertex u, find the *LeftAncestorHole* and *RightAncestorHole* that the left and right endpoints a_u and b_u lie in. If neither ancestor hole is $hole(u)$, find the highest priority source belonging to $hole(u)$. This will be the highest priority source in set A for u.
10. For every type_1 vertex u, use the source range trees for the *LeftAncestorHole* and *RightAncestorHole* found in the previous step to find the highest priority source belonging to each hole which lies in the interval (a_u, b_u). Of the two sources found, keep the higher priority source as the highest priority source in set D for u. Also find at the same time which child of the ancestor hole is on the path from $hole(u)$ to the ancestor hole. Let these be $left\_ancestor\_child(u)$ and $right\_ancestor\_child(u)$, abbreviated $lac(u)$ and $rac(u)$.
11. Calculate for each u the difference in level numbers for $hole(u)$ and $lac(u)$. The difference expressed in binary will give the set of $O(\log h)$ path segments, each of appropriate length 2^j, which are followed in going up the nesting tree from $hole(u)$ to $rac(u)$. Using the tables found for each hole in the nesting tree in procedure *BuildRHOPHoleTree*, follow the appropriate size pointers for each path segment, and in $O(\log h)$ steps, find the highest priority source along the *BestRightSources*

in the path from $hole(u)$ to $rac(u)$ in the nesting tree. This is the highest priority source in set B for u.

12. Repeat the previous step for the path from $hole(u)$ to $lac(u)$. This gives the highest priority source in set C for u.
13. Find for each u the highest priority source among the sources found in steps 9, 10, 11, and 12. Assign this source to u.

Again every step in the procedure can be run in either $O(\log m)$ or $O(\log n)$ time using $O(m)$ or $O(n)$ processors. Since for a planar graph G with no multi-edges or self-loops, the number of edges is $O(n)$, the whole procedure runs in $O(\log n)$ time with $O(n)$ processors on a CRCW PRAM.

4.4.6. Assigning sources to type_2 vertices

Once all type_1 vertices have been assigned sources, we can use Corollaries 4.13 and 4.14 to assign sources to type_2 vertices. If in a CS graph G, we cut all edges which are crossed by forward arcs, any connected component which results will be shown to have only one type_1 vertex in it. Suppose a type_2 vertex x belongs to a connected component with a type_1 vertex v_{top}. Then there exists no forward arc which encloses x but not v_{top} so that we can apply Corollaries 4.13 and 4.14.

Let x be associated with a back arc B with endpoints (c, d). Let the forward arc associated with v_{top} have endpoints (a, b). We will use both RHOPhole and RHIPhole structures. Let x belong to hole H_1 and c lie in hole G_1 in the RHOPhole structure. Then by Corollary 4.13, any source s_1 which belongs to an ancestor (not necessarily proper) of H_1 in the RHOPhole nesting tree and lies between a and c will have a path to x. Similarly, let x belong to hole H_2 and d lie in hole G_2 in the RHIPhole structure. By Corollary 4.14, any source s_2 which belongs to an ancestor (not necessarily proper) of H_2 in the RHIPhole nesting tree and lies between d and b will have a path to x. Moreover such sources s_1 and s_2 will be the only sources with paths to x. (Figure 47)

Thus to find the highest priority source with a path to x, we repeat the procedure *AssignType1Vertex* twice: once as applied to the RHOPhole structure using x as a type_1 vertex and (a, c) as its associated forward arc and once as applied to the RHIPhole structure using x as a type_1 vertex with (d, b) as its associated forward arc. Of the two sources found, the higher priority source is assigned to x. Only minor modifications to *AssignType1Vertex* are needed; we remove steps 5 and 6 which find the type_1 vertices and their associated forward arcs, and replace all mention of type_1 vertex u and its forward arc (a_u, b_u) with type_2 vertex x and the interval (a_x, c_x) or (d_x, b_x).

We now state the lemma that assures us that there will be only one type_1 vertex in each of the connected components we get when we cut a CS graph along all the forward arcs.

Lemma 4.19 *Let u and v be two different type_1 vertices. Then there will always exist a forward arc which encloses one but not the other.*

Proof. This is a proof by contradiction. Suppose there exists no forward arc which encloses one but not the other. Let u be associated with the forward arc A_1 and v with the forward arc A_2. Then A_1 must enclose v and hence all sources with paths to v, and

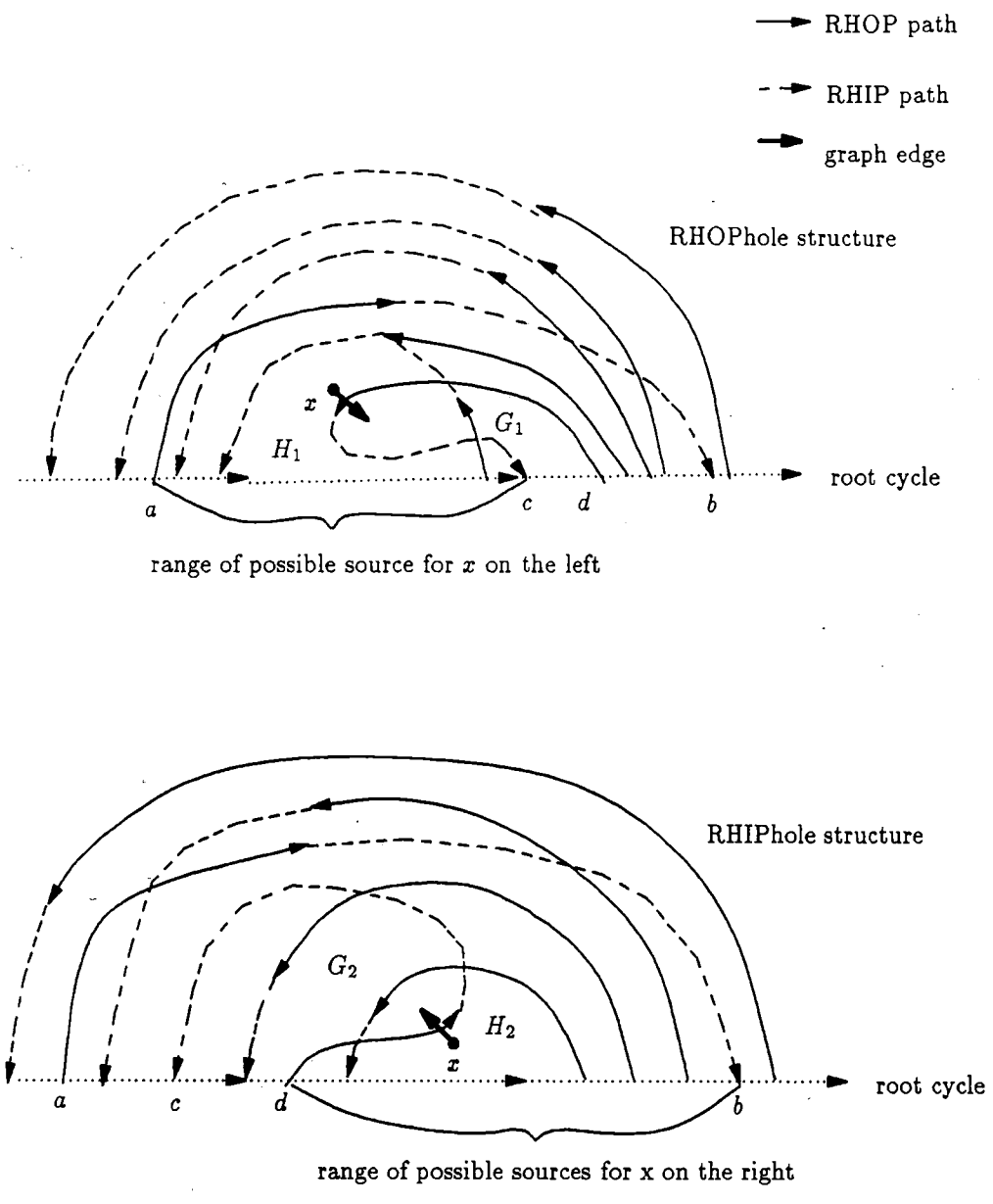


Figure 47
Type_2 vertex

A_2 must enclose u and hence all sources with paths to u. The only way this can occur is if both arcs share the same endpoints.

When two forward arcs share the same endpoints, either the paths of the two arcs cross or they do not. If they do not, one arc is inside the other, and the inside arc encloses one type_1 vertex, but not the other. If the paths cross, it can only be the RHOP path of one crossing the RHIP path of the other from the right. In this case A_1 encloses u but not v, and A_2 encloses v but not u. ∎

The following procedure will assign sources to type_2 vertices.

procedure *AssignType2Vertex*
(* This procedure takes a CS graph G and assigns for each type_2 vertex x in G the highest priority source with a path to x. *)
1. Run procedure *LabelForwardBack*.
2. Mark every type_1 vertex using information from *LabelForwardBack*.
3. Mark every type_2 vertex. Let the back arc associated with a type_2 vertex x be (c_x, d_x).
4. In a copy of G, cut all edges crossed by a *forward* edge. Find all connected components. For every type_2 vertex x, find the type_1 vertex v_x which is in the same connected component and the forward arc (a_x, b_x) associated with the v_x.
5. Run procedures *FindRHOPHoles*, *BuildRangeTreeForHoles*, *BuildRangeTreesForSources*, and *BuildRHOPHoleTree* using the RHOPhole structure.
6. Repeat steps 8 through 13 of procedure *AssignType1Vertex*, using x for the type_1 vertex and the endpoints a_x and c_x for the left and right endpoints of (a_u, b_u). Label the sources found as the $LeftSource(x)$.
7. Repeat step 5 for the RHIPhole structure.
8. Repeat step 6 using the RHIPhole structure instead of the RHOPhole structure, and using for every type_2 vertex x, the endpoints d_x and b_x as the left and right endpoints of the interval respectively. Label the sources found as $RightSource(x)$.
9. Assign the higher priority source of the $LeftSource(x)$ and the $RightSource(x)$ to the type_2 vertex x.

Again every step in the procedure and hence the entire procedure itself can be done in $O(\log n)$ time using $O(n)$ processors.

4.4.7. Assigning sources to type_3 vertices

The task of assigning sources to the remaining type_3 vertices is now a simple one. Again we apply Theorem 4.9. If we look at the connected components in the subgraph induced by the type_3 vertices, all in-edges to any one such connected component must be from type_1 or type_2 vertices. By the following lemma, we see that if a source s has a path to a type_1 or type_2 vertex bordering the connected component of type_3 vertices, it has a path to all type_3 vertices in the component. Thus if we find know the source assignments for all type_1 and type_2 vertices bordering the component, we select the highest priority source among them and assign it to all type_3 vertices in the component.

Lemma 4.20 *Let Comp be a connected component in the subgraph of a CS graph G induced by type_3 vertices. Let v be a type_1 or type_2 vertex with an out-edge to a vertex in Comp, and let s be a source with a path to v. Then if x is a type_3 vertex belonging to Comp, then there exists a path from s to x.*

Proof. (Figure 48) We will prove that there exists no forward arc enclosing x but not s or back arc enclosing s but not x. Then by Theorem 4.9, there exists a path from s to x.

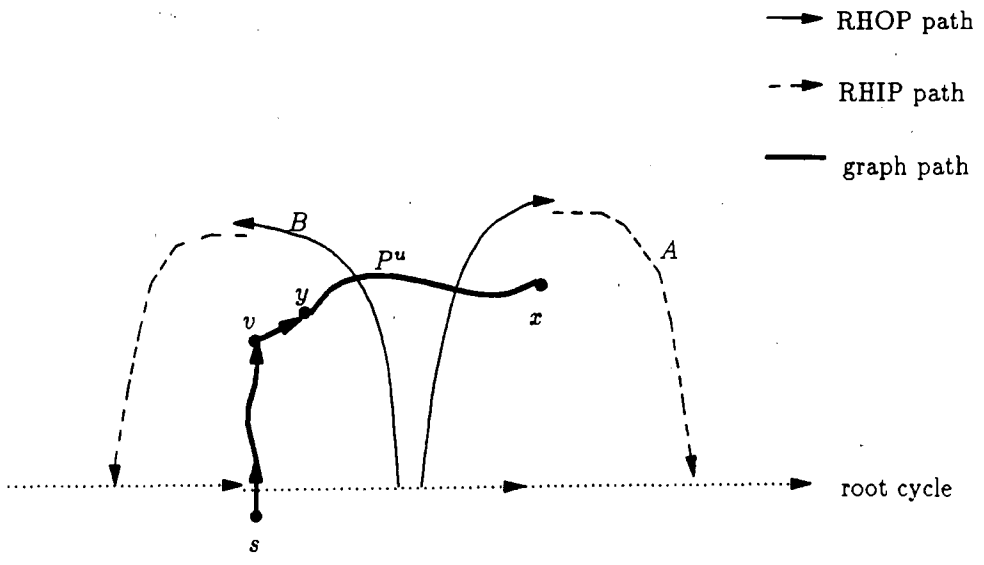


Figure 48
Illustration for Lemma 4.20

Let y be the type_3 vertex belonging to $Comp$ to which v has an out-edge. Since x and y belong to $Comp$, there must exist an undirected path P^u between x and y such that all vertices on P^u are type_3 vertices belonging to $Comp$.

Assume for a proof by contradiction that there exists a forward arc A which encloses x but not s. Then A cannot enclose y: if it did, there can be no path from s to y since s is outside A, but we know there is a path from s to y composed of the path from s to v and the out-edge from v to y. Since A encloses x but not y, some dual edge e^* of A must cross a graph edge e on P^u.

By Property 4.12 and its analogous property for RHIP structure, we know that all edges on an RHOP path from the root to a *forward* edge are *forward* edges, and that all edges on an RHIP path from a *forward* edge to the root are *forward* edges. Hence all edges on A must be *forward* edges, including e^*. This implies that some vertex on P^u between x and y is a type_1 vertex, giving a contradiction.

We give a similar proof by contradiction to show that there exists no back arc B enclosing s but not x. Again B must enclose y: if it did not, there can exist no path from s to y. Since B would then enclose y but not x, it must cross P^u at some edge e.

By Property 4.14 and its analogous property for the RHIP structure, all edges on a back arc must be either *back* edges or *forward* edges (i.e., none can be *bt* edges). Thus some vertex on P^u between x and y must be either a type_1 or type_2 vertex, giving the contradiction. ∎

The following procedure assigns sources to type_3 vertices.

procedure *AssignType3Vertex*

(* This procedure takes a CS graph G in which all type_1 and type_2 vertices have been assigned the highest priority sources which have paths to the vertices and assigns to every type_3 vertex x the highest priority source s with a path to x. *)

1. Find the connected components in the subgraph of G induced by type_3 vertices.
2. For every connected component $Comp$ found in the previous step, find type_1 and type_2 vertices with out-edges to $Comp$.
3. For every $Comp$, find the sources assigned to the type_1 and type_2 vertices found in the previous step. Include in this set of sources any source which has an out-edge to $Comp$. Find the highest priority source in each set of sources and assign it to all vertices in the associated $Comp$.

All steps can be done in $O(\log n)$ time using $O(n)$ processors in a CRCW PRAM model.

4.4.8. Assigning vertices inside a supersource boundary cycle

We now can write the algorithm which will assign sources to all vertices inside a supersource boundary cycle.

procedure *AssignSupersourceVertices*

(* Given a planar dag G all of whose sources are contiguous, so that the positive faces in the dual graph form a connected component, this procedure assigns each vertex to the highest priority source which has a path to the vertex.*)

1. Run procedure *CutSupersource*.
2. For each separate subgraph produced in step 1, do
   a. Run procedure *AssignType1Vertex*.
   b. Run procedure *AssignType2Vertex*.
   c. Run procedure *AssignType3Vertex*.

Since the total number of all edges and all vertices is $O(n)$ for the planar graph G, the procedure runs in $O(\log n)$ time using $O(n)$ processors. Thus by Theorem 4.8, Algorithm 5 for partitioning a multisource planar dag will run in $O(\log k \log n)$ time using $O(n)$ processors in the CRCW PRAM model.

4.5. Algorithm for depth first search of a planar dag

Given the algorithm for partitioning a multisource planar dag, the algorithm for depth first search is a simple one.

Algorithm 6. Depth first search of a planar dag

If the graph has only one source

then run the Algorithm 3 for the ET dfs of a single source planar dag

else

run Algorithm 5 for partitioning vertices among multisources,

run Algorithm 3 for the ET dfs of a single source planar dag on each subgraph.

Since Algorithm 5 requires $O(\log^2 n)$ time and $O(n)$ processors in a CRCW model, and Algorithm 3 requires $O(\log n)$ time and $O(n)$ processors in a EREW model, this algorithm has the same time and processor requirement as Algorithm 5.

CHAPTER 5
Conclusions

5.1. Summary

We have given several parallel depth first algorithms for planar directed acyclic graphs. For the case of a planar dag with a single source and a single sink, the source and sink not necessarily on the same face, we showed an optimal $O(\log n)$ time $O(n/\log n)$ algorithm in the EREW PRAM model. For the case of a planar dag with a single source, we gave a $O(\log n)$ time $O(n)$ processor EREW algorithm. In both these cases, the embedding is assumed to be given. For the general case of a planar dag with multiple sources, we gave a $O(\log^2 n)$ time $O(n)$ processor CRCW algorithm.

As part of the depth first search algorithm for planar dags, we found an algorithm for partitioning vertices in a multisource planar dag to the different sources and producing an ordering of the sources consistent with a depth first search ordering. A simpler variation of this partition algorithm gave a $O(\log^2)$ time $O(n)$ processor CRCW algorithm for the single source reachability problem. As a consequence, the linear processor depth first search algorithm of Kao and Klein for the directed planar graph will have its time requirement cut by a $\log^2 n$ factor to $O(\log^6 n)$. Given a single source planar dag, we also presented a $O(\log n)$ time $O(n)$ CRCW procedure to assign vertices to the highest priority out-edge of the source such that the vertex is reachable from the source by a path using that out-edge. The assignment of the priorities can be arbitrary.

This work introduces the concept of a planar Euler tour depth first search, a particular depth first search with properties which can be exploited in a parallel algorithm. Extensive use was made of two related structures which we called the Right Hand Out-Path structure and the Right Hand In-Path structure. Their properties may make them useful for other parallel algorithms for planar graphs.

5.2. Open questions

There are still many problems in the area of parallel search algorithms for directed graphs.

1. Although the time requirement for the depth first search of a planar graph has been brought down to $O(\log^6 n)$, it is still quite high. Can the time requirement be lowered to $O(\log^2 n)$ or lower, keeping the linear processor bound?
2. The breadth first search for planar graphs can be done in $O(\log^2 n)$ time using $O(n^{1.5})$ processors if a randomization is used [PR87, GM87]. Without randomization, the best results are for the general graphs using $O(M(n))$ processors and $O(\log^2 n)$ time, where $M(n)$ is the number of processors needed for matrix multiplication [GM88]. Is there a deterministic linear-processor NC algorithm for planar breadth first search?

3. Is there a deterministic NC algorithm for depth first search for the general undirected or directed case?

References

[AA87] Alok Aggarwal and Richard J. Anderson. A random NC algorithm for depth first search. In *Proc. 19th Annual Symposium on Theory of Computing*, pages 325–334, 1987.

[AAK89] A. Aggarwal, R.J. Anderson, and M.Y. Kao. Parallel depth-first search in general directed graphs. In *21st Symposium on Theory of Computing*, pages 297–308, 1989.

[AHU74] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The design and analysis of computer algorithms.* Addison-Wesley Pub. Co., Reading, Mass., 1974.

[AKS78] Miklós Ajtai, János Komlós, and Endre Szemeredédi. There is no fast single hashing algorithm. *Information Processing Letters*, 7(6):270–273, October 1978.

[AM88] Richard J. Anderson and Gary L. Miller. Deterministic parallel list ranking. In J. H. Reif, editor, *VLSI Algorithms and Architectures*, pages 81–90, Corfu, Greece, June/July 1988. Springer-Verlag.

[BB78] R. Creighton Buck and Ellen F. Buck. *Advanced Calculus.* International Series in Pure and Applied Mathematics. McGraw-Hill, New York, third edition, 1978.

[BB88] Gilles Brassard and Paul Bratley. *Algorithmics: Theory and Practice.* Prentice-Hall, Englewood Cliffs, NJ, 1988.

[BBS90] Béla Bollabás, Andrei Z. Broder, and Istvan Simon. The cost distribution of clustering in random probing. *Journal of the ACM*, 37:224–237, 1990.

[Coo71] S. A. Cook. The complexity of theorem proving procedures. In *Proc. Third Annual ACM Symposium on Theory of Computing*, pages 151–158, 1971.

[CV88] Richard Cole and Uzi Vishkin. Optimal parallel algorithms for expression tree evaluation and list ranking. pages 91–100, New York, 1988. Springer-Verlag. Lecture Notes in Computer Science, No. 319.

[Eve79] Shimon Even. *Graph Algorithms.* Computer Science Press, Inc., Potomac, Maryland, 1979.

[GM87] Hillel Gazit and Gary L. Miller. A parallel algorithm for finding a separator in planar graphs. In *28th Annual Symposium on Foundations of Computer Science*, pages 238–248, 1987.

[GM88] Hillel Gazit and Gary L. Miller. An improved parallel algorithm for bfs of a directed graph. *Information Processing Letters*, 28(2):61–65, 1988.

[GPS87] Andrew V. Goldberg, Serge A. Plotkin, and Gregory E. Shannon. Parallel symmetry-breaking in sparse graphs. In *19th Annual ACM Symposium on Theory of Computing*, pages 315–324, New York, NY, May 1987. ACM, ACM.

[GS78] L. J. Guibas and E. Szemerédi. The analysis of double hashing. *Journal of Computer and System Sciences*, 16:226–274, 1978.

[Gui87] Leo J. Guibas. private communications, Fall 1987.

[Hag88] Torben Hagerup. Optimal parallel algorithms on planar graphs. In J. H. Reif, editor, *VLSI Algorithms and Architectures*, pages 24–32. Springer-Verlag, June/July 1988.

[Har72] Frank Harary. *Graph Theory.* Addison-Wesley, 1972.

[Hoe63] W. Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 58:13–30, 1963.

[HY88] Xin He and Yaacov Yesha. A nearly optimal parallel algorithm for constructing depth first spanning trees in planar graphs. *SIAM Journal on Computing*, 17(3):486–491, June 1988.

[JK88] J. Ja'Ja and S. Kosaraju. Parallel algorithms for planar graph isomorphism and related problems. *IEEE Transactions on Circuits and Systems*, 35(3):304–311, March 1988.

[Kao88] Ming-Yang Kao. All graphs have cycle separators and planar directed depth-first search is in DNC. In J. H. Reif, editor, *VLSI Algorithms and Architectures*, pages 53–63. Springer-Verlag, June/July 1988.

[Kar72] Richard M. Karp. Reducibility among combinatorial problems. In Raymond E. Miller and James W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, New York, 1972. The IBM Research Symposia Series.

[Kar86] Richard M. Karp. Combinatorics, complexity, and randomness. *Communications of the ACM*, 29(2):98–109, February 1986.

[KK82] Narendra Karmarkar and Richard M. Karp. The differencing method of set partitioning. Technical Report UCB/CSD 82/113, Computer Science Division (EECS), University of California, Berkeley, December 1982.

[KK90] M.Y. Kao and P.N. Klein. Toward overcoming the transitive closure bottleneck: Efficient parallel algorithms for planar digraphs. In *21st Annual ACM Symposium on Theory of Computing*, pages 181–192, 1990.

[Knu73] Donald E. Knuth. *Sorting and Searching*, volume III of *The Art of Computer Programming.* Addison-Wesley, Reading, MA, 1973.

[Kom86] János Komlós. private communications, 1986.

[KR88] Richard M. Karp and Vijaya Ramachandran. A survey of parallel algorithms for shared-memory machines. Technical Report UCB/CSD 88/408, Computer Science Division, University of California, Berkeley, CA 94720, March 1988. To appear in the *Handbook of Theoretical Computer Science* by North-Holland.

[KS90] Ming-Yang Kao and Gregory E. Shannon. Linear-processor NC algorithms for planar directed graphs II: Directed spanning trees. Technical Report 306, Computer Science Department, Indiana University, March 1990.

[LF80] R.E. Ladner and M.J. Fischer. Parallel prefix computation. *Journal of the ACM*, 27:831–838, 1980.

[LM88] George S. Lueker and Mariko Molodowitch. More analysis of double hashing. In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing*, pages 354–359, Chicago, IL, May 1988.

[Pip79] Nicholas Pippenger. On simultaneous resource bounds. In *20th Symposium on Foundations of Computer Science*, pages 307–311, 1979.

[Pip88] Nicholas Pippenger. private communications, January 1988.

[PR87] Victor Pan and John Reif. Fast and efficient solution of the path algebra problems. Technical Report 3, Computer Science Department, State University of New York at Albany, 1987. (Cited in Kao and Shannon's tech rep).

[Rei85] John H. Reif. Depth-first search is inherently sequential. *Information Processing Letters*, 20:229–234, 12 June 1985.

[RR89] V. Ramachandran and J. Reif. An optimal parallel algorithm for graph planarity. In *30th Annual Symposium on Foundations of Computer Science*, pages 282–287. IEEE Computer Society Press, October 1989.

[Sha88] Gregory E. Shannon. A linear-processor algorithm for depth-first search in planar graphs. *Information Processing Letters*, 29:119–123, October 1988.

[Smi86] J. R. Smith. Parallel algorithms for depth-first searches: I. planar graphs. *SIAM Journal on Computing*, 15(3):814–830, August 1986.

[SS90] Jeanette P. Schmidt and Alan Siegel. On aspects of the universality and performance for closed hashing. In *21st Annual ACM Symposium on Theory of Computing*, pages 355–366, 1990.

[TV85] Robert E. Tarjan and Uzi Vishkin. Finding biconnected components and computing tree functions in logarithmic parallel time. *SIAM Journal on Computing*, 14(4):862–874, November 1985.

[Ull72] Jeffrey D. Ullman. A note on the efficiency of hash functions. *Journal of the ACM*, 19:569–575, 1972.

[Wyl81] J. C. Wyllie. *The Complexity of Parallel Computation.* PhD thesis, Cornell University, Ithaca, NY, 1981.

[Yao85] Andrew C. Yao. Uniform hashing is optimal. *Journal of the ACM*, 32(3):687–693, July 1985.