All Graphs Have Cycle Separators and Depth-First Search on Planar Digraphs is in DNC

By

Ming-Yang Kao

Computer Science Department
Indiana University
Bloomington, IN 47405

TECHNICAL REPORT NO. 244

All Graphs Have Cycle Separators and Depth-First Search on Planar Digraphs is in DNC

by

Ming-Yang Kao

March, 1988

This work was supported in part by a 1987 Summer Faculty Fellowship from Indiana University at Bloomington.

All Graphs Have Cycle Separators

and

Depth-First Search on Planar Digraphs is in DNC*

Ming-Yang Kao

Department of Computer Science

Indiana University, Bloomington, IN 47405

Kao@Indiana.Edu

March 22, 1988

Abstract

All graphs have cycle separators. In sequential computation, a cycle separator can be found in $O(n+e)$ time for any undirected graph of n vertices and e edges; in $O((n+e)\log n)$ time for any directed graph. In parallel computation, finding a cycle separator is deterministic NC-equivalent to finding a depth-first search forest. In particular, finding a cycle separator for any *planar directed* graph is in *deterministic* NC; consequently, finding a depth-first search forest in any planar directed graph is in deterministic NC, too.

---

*This work was supported in part by a 1987 Summer Faculty Fellowship from Indiana University at Bloomington.

1 Introduction

A *separator* of a vertex-weighted digraph G is a set S of vertices or edges such that $G - S$ has no strongly connected component heavier than a constant fraction of the total weight of G. This definition also applies to undirected graphs if each undirected edge is replaced by a pair of directed edges [LT79] [Lei80] [Mil84]. A *cycle separator* is a vertex-simple directed cycle such that its vertices form a separator. (A single vertex is regarded as a trivial cycle.) Finding graph separators is a very important part of the divide-and-conquer strategies for a wide range of graph theoretical problems and applications. Many applications require *small* separators [LRT79] [Lei80] [Val81] [JV83] [FJ86] [PR87]. Others can use separators of *any size*; for example, Smith [Smi86] shows that cycle separators of any size can be used to parallelize depth-first search on *planar undirected* graphs. In fact, this paper concentrates on the relationship between depth-first search and cycle separators of any size for *general directed* graphs.

Miller [Mil84] shows that *planar undirected* graphs have *small* cycle separators; these graphs are the only non-trivial class previously known to have cycle separators. In contrast, this paper shows that all *general directed* graphs have cycle separators of *any size*. Not all graphs, however, can have small cycle separators; complete graphs are obvious examples. The proof of the universal existence of cycle separators is based on depth-first search; consequently, the proof has many complexity implications on finding depth-first search trees and finding cycle separators for digraphs. These implications and other contributions are briefly presented now; in the following let n be the number of vertices of any given graph, and let e be the number of edges.

1. Sequential Complexity

   - For *planar undirected* graphs, Miller [Mil84] gives a linear-time algorithm to find *small* cycle separators.

- This paper shows that for *general undirected* graphs, cycle separators can be found in linear time, and that for *general directed* graphs, cycle separators can be found in $O((n+e)\cdot\log n)$ time.

2. Parallel Complexity

   (a) Deterministic NC-Equivalence

   This paper shows that finding a cycle separator and finding a depth-first search tree are *deterministically NC-equivalent.*

   (b) Undirected Graphs

   - For *planar* undirected graphs, Gazit and Miller [GM87] present an efficient RNC algorithm for finding cycle separators of *small size*; Goldberg, Plotkin, and Shannon [GPS87] also show that cycle separators of *any size* can be found *deterministically* in $O(\log n)$ time and a linear number of processors on a CRCW PRAM.
   - For *general* undirected graphs, Reif shows that *ordered* depth-first search is P-complete [Rei85]. Aggarwal and Anderson [AA87] show that *un-ordered* depth-first search is in RNC; this paper shows that immediately from the DNC-equivalence, finding a cycle separator is also in RNC.

   (c) Directed Graphs

   - For *acyclic* digraphs, Ghosh and Bhattacharjee [GB84] give a deterministic NC algorithm for lexicographically-first depth-first search.
   - For *planar* digraphs, this paper shows that finding cycle separators is in *deterministic* NC; immediately from the DNC-equivalence, depth-first search is also in *deterministic* NC. The algorithms are non-trivial.
   - For *general* digraphs, Schevon and Vitter [SV85] give a deterministic NC algorithm to

test whether any given spanning forest is a DFS forest. The algorithm also produces a DFS order if the forest is a DFS forest.

- For *general* digraphs, the author also discovered that cycle separators and thus depth-first search forest can be computed probabilistically in $O(\sqrt{n}\log^c n)$ time and a polynomial number of processors; this result is obtained by combining the ideas from the proof for the universal existence of cycle separators and from a randomized parallel undirected depth-first search of the same complexity by Anderson [And85]. This result is omited from this paper because recently the author, Aggarwal, and Anderson have jointly discovered an RNC algorithm for DFS on *general* digraphs. This new result will be reported in another paper.

This paper is organized as follows. Section 2 gives the DFS-based proof for the universal existence of cycle separators; this section also discusses the immediate algorithmic implications of the proof in sequential and parallel computation. Section 3 uses cycle separators to build in parallel a DFS forest for any general digraph; this section also proves the deterministic NC equivalence between performing DFS and finding cycle separators. Section 4 describes the DNC algorithm for finding a cycle separator in any planar digraph; from the DNC equivalence, this cycle separator algorithm yields a DNC algorithm for DFS on planar digraphs. Section 5 gives the conclusions.

2 The Existence Proof of Cycle Separators

Subsection 2.1 gives the DFS-based proof for the universal existence of cycle separators. Subsection 2.2 discusses the immediate algorithmic implications of the proof in sequential and parallel computation.

2.1 The Proof

Throughout this paper a weighted graph always refers to one with non-negative vertex weights; to avoid triviality, at least one vertex weight is greater than zero. Let G be such a weighted digraph. Let $W(v)$ denote the weight of any vertex v; let $W(K)$ denote the total weight of any vertex set K. K is *overweight* if $W(K) > W(G)/2$.

Definition 1 (*Generic Separators*)

Let G be any weighted digraph. A separator S of G is a vertex set of G such that $G - S$ has no overweight strongly connected component.

A *cycle* (or *path*) separator is a vertex-simple directed cycle (or path, respectively) such that its vertices together form a separator. This section offers two separator theorems: *all weighted digraphs have path separators and cycle separators.* The theorems can be immediately applied to any weighted *undirected* graph by substituting each undirected edge with a pair of directed edges; in such a substitution, any directed cycle separator obtained in the theorems cannot degenerate into an undirected edge because it is one of the following three cases: the cycle of the empty set, the cycle of a single vertex, or a cycle of at least three vertices.

In the following, depth-first search is used to identify a path separator for any given weighted digraph; then the path separator is converted into a cycle separator.

Theorem 2 (*Path Separator*)

Every weighted digraph has a path separator.

Proof. Let G be any given weighted digraph with n vertices. Assume w.l.o.g. that G is strongly connected. Otherwise replace G with its maximum-weight strongly connected component G' because every path separator for G' is also one for G.

Construct a path separator for G as follows. Apply depth-first search (DFS) to G from an arbitrary vertex r and list the vertices in the DFS last-visit order $x_1, ..., x_n$. Let f be the smallest index such that $1 \leq f \leq n$ and $W(x_1) + \cdots + W(x_f) \geq W(G)/2$. Then $W(x_1) + \cdots + W(x_{f-1}) < W(G)/2$ and $W(x_{f+1}) + \cdots + W(x_n) \leq W(G)/2$. Now let P be the vertex-simple directed path in the DFS tree from the root r to x_f.

Prove that P is a path separator of G as follows. Let $A = \{x_1, ..., x_{f-1}\}$ and $B = G - (A \cup P)$. Arrange the DFS tree in such a way that for all last-visit numbers i and j, if $i > j$, the vertex x_i is either above or to the right of the vertex x_j in the tree. In such an arrangement, the vertex x_f is above or to the right of every vertex in A because the index f is greater than the last-visit numbers of all vertices in A. Moreover, every vertex in B is to the right of x_f because the last-visit numbers of vertices in B are all greater than f and because P consists of x_f and all its ancestors. The above two observations imply that every vertex in B is to the right of every vertex in A. Because all cross edges in DFS are directed edges from the right to the left, there are no undirected edges between A and B, and no directed edges from A to B. Therefore, any strongly connected component of $G - P$ is contained either entirely in A or entirely in B. Consequently, the total weight of any strongly connected component of $G - P$ is no more than $W(A)$ or $W(B)$, which both are in turn no more than $W(G)/2$ because $A = \{x_1, ..., x_{f-1}\}$ and $B \subseteq \{x_{f+1}, ..., x_n\}$. ∎

Theorem 3 (*Cycle Separator*)

Every weighted digraph has a cycle separator.

Proof. Let G be any given weighted digraph; let $P = u_1, ..., u_r$ be a path separator of G obtained from the Path Separator Theorem. Convert P to a cycle separator as follows. Let t be the largest index such that $1 \leq t \leq r$ and $G - \{u_1, ..., u_{t-1}\}$ has an overweight strongly connected component X; then the set X must contain u_t because the path $u_1, ..., u_t$ is still a separator. If such an index t does not exist, then the empty set is a trivial cycle separator; otherwise continue the conversion

and let $P' = u_1, ..., u_t$. Let s be the smallest index such that $1 \leq s \leq t$ and $G - \{u_{s+1}, ..., u_t\}$ has an overweight strongly connected component Y; then the set Y must contain u_s because the path $u_s, ..., u_t$ is still a separator. To finish the conversion, there are two cases based on whether or not $s = t$. If $s = t$, then the vertex u_s is a trivial cycle separator. If $s \neq t$, then $u_s \notin X$ and $u_t \notin Y$. Furthermore, because X and Y are overweight and strongly connected, $X \cap Y$ must have a vertex w such that there is a vertex-simple directed path Q from u_t through X to w and then from w through Y to u_s. Finally, because $X \cup Y$ does not contain any of $u_{s+1}, ..., u_{t-1}$, the path Q and the path $u_s, ..., u_t$ form a vertex-simple directed cycle of at least three vertices; this cycle is a separator because the path $u_s, ..., u_t$ is a separator. ■

2.2 The Immediate Algorithmic Implications

In the following, let n be the number of vertices and e be the number of edges in any given graph.

1. **Compute Path and Cycle Separators Sequentially**

   - **Path Separators.** The proof of the Path Separator Theorem yields a simple linear-time algorithm for finding a path separator of any weighted digraph.
   - **Cycle Separators.** The proof of the Cycle Separator Theorem can be easily translated into algorithms for converting a path separator into a cycle separator. In general, such a conversion takes $O((n+e) \cdot \log n)$ time if bisection is used to find the indices t and s; in the case of undirected graphs, the indices t and s can easily be found in linear time. Therefore, it takes $O(n + e)$ time to find a cycle separator for any weighted undirected graph, and $O((n + e) \cdot \log n)$ time for any weighted digraph.

2. **Use DFS to Find a Path Separator in Parallel**

For any weighted digraph, any path separator of its maximum weight strongly connected component is also a path separator of the digraph itself; moreover, computing a maximum weight strongly connected component is in NC. Consequently, the following discussion may assume w.l.o.g. that the given digraph is strongly connected.

Algorithm 4 (*Find-a-Path-Separator-from-a-DFS-Tree*)

Input: *a weighted strongly connected digraph G of n vertices, and a DFS tree T of G.*

Output: *a path separator of G.*

Ideas. The proof of the Path Separator Theorem indicates that in any DFS tree, at least one tree path is a path separator; therefore a path separator can be found by simultaneously testing all tree paths from the root to the leaves. Furthermore, if the last-visit order of the DFS tree is known, the proof explicitly gives a path separator by identifying the vertex x_f, which can be found by a standard prefix computation algorithm in $O(\log n)$ time and $O(n)$ processors on an EREW PRAM [KRS85].

3. Convert a Path Separator to a Cycle Separator in Parallel

Algorithm 5 (*Convert-a-Path-Separator-to-a-Cycle-Separator*)

Input: *a weighted digraph G of n vertices, and a path separator of G.*

Output: *a cycle separator of G.*

Ideas. The proof of the Cycle Separator Theorem can be implemented as follows. The index t can be found by bisecting and testing the path separator; after t is obtained, the index s can be found in the same way. The bisection process requires $O(\log n)$ tests of whether a path is a separator. To save time over the processor complexity, the index t can be found by simultaneously trying all possibilities; after t is obtained, the index s is treated in the same way.

3 Use Cycle Separators to Conduct Parallel DFS

The next theorem is the main result of this section.

Theorem 6 (*The DNC Equivalence between Conducting DFS and Finding Separators*)
Given an oracle for any of the following five problems, there is a deterministic NC algorithm for each of the other four problems.

1. *Conducting depth-first search on any digraph.*
2. *Finding a path separator of any weighted digraph.*
3. *Finding a path separator of any weighted digraph with unit vertex weights.*
4. *Finding a cycle separator of any weighted digraph.*
5. *Finding a cycle separator of any weighted digraph with unit vertex weights.*

Proof. The following six reductions suffice: $1 \leq 5$ (see the following overview), $5 \leq 4$ (obvious), $4 \leq 2$ (Algorithm 5), $2 \leq 1$ (Algorithm 4), $5 \leq 3$ (similar to $4 \leq 2$), and $3 \leq 2$ (obvious). ∎

The overview for the reduction from 1 to 5. A digraph is *rooted* at a vertex if the vertex is reachable to all other vertices through directed paths. A *partial* DFS tree in a rooted digraph is a subtree of a DFS tree such that the digraph and both trees are rooted at the same given vertex. Subsection 3.1 uses cycle separators to extend any partial DFS tree of any rooted digraph. Subsection 3.2 applies this tree extending algorithm to build a DFS tree for any rooted digraph. Subsection 3.3 builds a DFS forest for any general digraph by decomposing the general digraph into rooted digraphs.

3.1 Use Cycle Separators to Extend a Partial DFS Tree

Let G be any given rooted digraph. Let T be any partial DFS tree of G; let $x_1, x_2, ..., x_t$ be the vertices in T listed in the DFS *last-visit* order. Further let $y_{i,1}, ...y_{i,k_i}$ be the vertices not in T but incident

from x_i; the order of $y_{i,1}, ...y_{i,k_i}$ is *arbitrary*. Also for any two integer pairs $p = (i, j)$ and $p' = (i', j')$, let $p < p'$ denote that either $i < i'$ or ($i = i'$ and $j < j'$). For any digraph D and its vertex x, let $R(x, D)$ denote the set of vertices reachable from x through directed paths.

Definition 7 (*Dangling Subgraphs*)

Let $DSG(i,j) = R(y_{i,j}, G-T) - \cup_{p<(i,j)} R(y_p, G-T)$. *The non-empty* $DSG(i,j)$*'s are actually digraphs rooted at their corresponding* $y_{i,j}$*'s, and are called the dangling subgraphs induced by* T.

Lemma 8

The following three items form a DFS tree for G*:*

1. *the partial DFS tree* T,
2. *the edge set* $\{x_i \rightarrow y_{i,j} \mid DSG(i,j) \neq \emptyset.\}$, *and*
3. *an arbitrary DFS tree rooted at* $y_{i,j}$ *for each non-empty* $DSG(i,j)$.

Proof. Finish the partial DFS tree T into the $DSG(i,j)$'s by visiting the vertices $y_{i,j}$'s according to the lexicographic order on their indices. ∎

The lemma provides a parallel computation scheme to build a DFS tree because the dangling subgraphs are pairwise disjoint and their DFS trees can be computed independently. For the purpose of recursion, a dangling subgraph is *good* if it has at most 1/2 of the vertices in G; otherwise the subgraph is *bad*. A partial DFS tree is *good* if all its dangling subgraphs are good; otherwise the tree is *bad*. A bad partial DFS tree has exactly one bad dangling subgraph because the dangling subgraphs are pairwise disjoint. To obtain a good partial DFS tree, a bad partial DFS tree can be iteratively extended to break its bad dangling subgraph; the progress on breaking the bad dangling subgraph is measured by the size of *some* strongly connected components described as follows. Let T be a bad partial DFS tree for G; let $DSG(i_0, j_0)$ be its bad dangling subgraph. Further let H be the induced digraph of $DSG(i_0, j_0)$ condensed at its strongly connected components. Because H is acyclic and

because $DSG(i_0, j_0)$ is bad and rooted, there is a vertex in H such that this vertex and its descendants together consist of more than $1/2$ of the vertices in G.

Definition 9 (*Lowest Extension Ends*)

An extension end of T is a vertex in H such that this vertex and its descendants together consist of more than $1/2$ of the vertices in G. A lowest extension end is one such that none of its children is an extension end, or equivalently none of its descendants is an extension end.

Algorithm 10 (*Extend-a-Partial-DFS-Tree-to-Cut-Extension-Ends*)

Input: *a rooted digraph G, and a bad partial DFS tree T of G.*

Output: *a partial DFS tree T' of G such that either T' is good or every extension end of T' consists of at most $1/2$ of the vertices of the same extension end of T.*

Oracle: *find a cycle separator for any given digraph with unit vertex weights.*

Methods. First find a lowest extension end L of T. Then find a cycle separator C of L. Then find a vertex-simple directed path P in $DSG(i_0, j_0)$ from y_{i_0,j_0} to an arbitrary vertex z in C such that P and C share only z. Then let C' be the path consisting of C without the edge incident *to* z. Finally, let $T' = T \cup \{x_{i_0} \rightarrow y_{i_0,j_0}\} \cup P \cup C'$.

Correctness. T' is a partial DFS tree of G because $P \cup C'$ is a vertex-simple directed path rooted at y_{i_0,j_0} in $DSG(i_0, j_0)$. Furthermore, if T' is still bad, then every extension end of T' is a strongly connected component of $L - (P \cup C')$; consequently every extension end of T' consists of at most $1/2$ of the vertices of the same extension end L of T.

Complexity. The following discussion shows that the algorithm is in deterministic NC. The $DSG(i, j)$'s can be obtained by computing the transitive closure of $G - T$ and by using the fact that $u \in DSG(i, j)$ if and only if (i, j) is the smallest index such that u is reachable from $y_{i,j}$ in $G - T$. After the $DSG(i, j)$'s are found, the bad dangling subgraph $DSG(i_0, j_0)$ can be easily identified.

Furthermore, the induced digraph H of $DSG(i_0, j_0)$ can be obtained by computing the transitive closure of $DSG(i_0, j_0)$; from the transitive closure of $DSG(i_0, j_0)$, finding the lowest extension end L is easily in deterministic NC. With L found, the cycle separator C of L can be obtained by a call of the given oracle; then the path P can be efficiently computed by the NC Bread-First Search algorithm by Gazit and Miller [GM87].

3.2 Build a DFS Tree for Any Rooted Digraph

The next algorithm iteratively applies Extend-a-Partial-DFS-Tree-to-Cut-Extension-Ends to extend the root of any rooted digraph into a DFS tree.

Algorithm 11 (*Build-a-DFS-Tree-for-a-Rooted-Digraph*)

Input: *a digraph G rooted at r.*

Output: *a DFS tree rooted at r for G.*

Oracle: *Extend-a-Partial-DFS-Tree-to-Cut-Extension-Ends.*

Method and Complexity. Initially, let T be the tree consisting of only r. Then iteratively use Extend-a-Partial-DFS-Tree-to-Cut-Extension-Ends to extend T into a good partial DFS tree. Because the tree extending oracle cuts by half the size of some lowest extension end, it takes $O(\log n)$ such iterations to obtain a good partial DFS tree. Once this good tree is found, compute its dangling subgraphs; as discussed before, finding the dangling subgraphs is in NC. Finally, recurse the above computation on the dangling subgraphs; the depth of the recursion is $O(\log n)$. The algorithm is clearly in NC.

3.3 Build a DFS Forest for Any General Digraph

The next algorithm uses Build-a-DFS-Tree-for-a-Rooted-Digraph to build a DFS forest for any general digraph by decomposing the general digraph into rooted digraphs.

Algorithm 12 (*Build-a-DFS-Forest-for-a-General-Digraph*)

Input: *a general digraph G.*

Output: *a DFS forest for G.*

Oracle: *Build-a-DFS-Tree-for-a-Rooted-Digraph.*

Ideas and Complexity. First decompose G into rooted digraphs as follows such that a DFS forest for G can be obtained by taking an *arbitrary* DFS tree from each of the rooted digraphs. List the vertices of G in an *arbitrary* order $x_1, ..., x_n$; then let $OWN(x_i) = R(x_i, G) - \cup_{p<i} R(x_p, G)$. The non-empty $OWN(x_i)$'s are digraphs rooted at their corresponding x_i's; furthermore, these rooted digraphs satisfy the desired decomposition property, and can be found in the same way as the dangling components are obtained. Once these rooted digraphs are obtained, apply Build-a-DFS-Tree-for-a-Rooted-Digraph to compute their DFS trees and thus build a DFS forest for G. The algorithm is clearly in NC.

4 A Deterministic NC Depth-First Search on Planar Digraphs

The main result of this section is the next theorem.

Theorem 13

Finding a cycle separator for any weighted planar digraph is in deterministic NC; from the NC-Equivalence Theorem, building a DFS forest for any planar digraph is also in deterministic NC.

The theorem is proven by the following description of a cycle separator algorithm for any weighted planar digraph. The complexity of the cycle separator algorithm is $O(\log^5 n)$ time and $O(n^4)$ processors on an EREW PRAM for any graph of n vertices; the complexity for the corresponding DFS algorithm is $O(\log^7 n)$ time and $O(n^4)$ processors. Such complexity is beyond practical application and is not worth the space for a tedious straightforward analysis; thus the following discussion con-

centrates on showing that the cycle separator algorithm is in deterministic NC, and omits the analysis for its complexity. For ease of understanding, the cycle separator algorithm and its subroutines are presented in a top-to-bottom hierarchical way. Subsection 4.1 gives an overview of the cycle separator algorithm; the subsequent subsections discuss in detail the subroutines used in the algorithm.

To shorten terminologies, all cycles and paths in the following discussion refer to vertex-simple directed ones unless explicitly stated otherwise.

4.1 An Overview of the Cycle Separator Algorithm

Algorithm 14 (*Find-a-Cycle-Separator*)

Input: *any weighted planar digraph G of n vertices.*

Output: *a cycle separator of G.*

Overview. A vertex set is called a *block* if it is strongly connected in G. A *path-block separator* is a disjoint pair of a path and a block whose vertices form a separator. G has an obvious path-block separator, namely, the empty path and its maximum-weight strongly connected component. The cycle-separator algorithm starts with this trivial separator and iteratively cuts by half the number of vertices in the block while building up the path. After $O(\log n)$ cuts, the block is left with at most one vertex. The cycle separator algorithm finishes by merging this one-vertex block and the final path into a cycle separator, using the same idea as in the proof of the Cycle Separator Theorem.

The block-cutting procedure is described as follows. Let (P_{in}, B_{in}) be any path-block separator; the goal is to find another path-block separator (P_{out}, B_{out}) such that $| B_{out} | \leq | B_{in} | /2$, where $| \cdot |$ denotes the number of vertices in a vertex set.

Step 1. To break B_{in} into smaller pieces, first use the NC algorithm by Lovász [Lov85] to compute an ear decomposition $E_1, ..., E_k$ for B_{in}. Then find the *smallest* index f such that there are at least $| B_{in} | /2$ vertices in $E_1 \cup \cdots \cup E_f$. Let $B_{big} = E_1 \cup \cdots \cup E_f$; also let $B_{small} = E_1 \cup \cdots \cup E_{f-1}$. From

the property of the ear decompositions, B_{big} and B_{small} are blocks; from the choice of the index f, $\mid B_{big} \mid \geq \mid B_{in} \mid /2$ but $\mid B_{small} \mid \leq \mid B_{in} \mid /2$. Now compute the strongly connected components $SC_1, ..., SC_s$ of $B_{in} - B_{big}$; the SC_i's are blocks such that $\mid SC_i \mid \leq \mid B_{in} \mid /2$. The structure of B_{in} can be simplified by treating B_{big} and the SC_i's as individual vertices; let $\overline{G}$ and $\overline{B_{in}}$ be the condensed versions of G and B_{in} respectively. Then in $\overline{B_{in}}$, all non-trivial cycles go through the same vertex B_{big}; a *daisy* will refer to any strongly connected planar digraph with such a centralization property. A *path-daisy separator* is one consisting of a disjoint pair of a path and a daisy; the result of this step is the path-daisy separator $(P_{in}, \overline{B_{in}})$ for $\overline{G}$.

Step 2. (Subsections 4.4 and 4.5.) This step starts with $(P_{in}, \overline{B_{in}})$, and operates in $\overline{G}$. The daisy $\overline{B_{in}}$ can be thought of as a collection of cycles centered at the vertex B_{big}; the goal is to iteratively cut by half the number of cycles in $\overline{B_{in}}$ while building up P_{in}. However, there may be an exponential number of cycles in $\overline{B_{in}}$. For this concern, a novel approach is invented to cut $\overline{B_{in}}$ by cutting the so-called petals; in a planar digraph, a *petal bounded by an edge e* is a cycle of a maximal interior among the cycles going through e. In any *daisy*, there are at most two petals *bounded by* each edge; therefore after $O(\log n)$ cuts, $\overline{B_{in}}$ is left with at most one cycle. Then this one-cycle daisy and the final path in the separator are merged into a cycle separator C of $\overline{G}$, using the same idea as in the proof of the Cycle Separator Theorem; the result of this step is the cycle separator C of $\overline{G}$.

Setp 3. (Subsections 4.2 and 4.3.) This step starts with C and operates in G. A *pseudo-cycle* of G will refer to a cycle in any condensed version of G; C is a pseudo-cycle in G. A pseudo-cycle separator of G is one consisting of a pseudo-cycle; C is a pseudo-cycle separator of G. A pseudo-cycle of G is in fact a cycle of G attached with disjoint blocks; C may have more than one block. Again a novel approach is used to iteratively cut by half the number of blocks in C while keeping C a separator; after $O(\log n)$ cuts, the result of this step is a pseudo-cycle separator of at most one block.

Step 4. The one-block pseudo-cycle separator can be decomposed into a path and a block; the remaining block is either B_{big} or some SC_i. In the former case, B_{big} can be further decomposed into the ear E_f and the block B_{small}. Therefore the separator consists of two or three paths and a block at most half the size of B_{in}. To finish the block-cutting procedure, merge the paths in the decomposition into a single path, using the same idea as in the proof of the Cycle Separator Theorem; the resulting path-block separator is of the required block size.

4.2 Basics of Pseudo-Cycle Separators

- Let $G = (V, E)$. A *block* B of G is a non-empty vertex set of G such that the digraph $(B, E \cap (B \times B))$ is strongly connected. If $B_1, ..., B_k$ are pairwise disjoint blocks of G, then the *condensed graph* $\overline{G}$ of G at $B_1, ..., B_k$ is a weighted *planar* digraph defined in the following expected way: (1) Each B_i is treated as a vertex in $\overline{G}$; each vertex in G but not in any B_i remains a vertex in $\overline{G}$. (2) For any vertices x and y in $\overline{G}$, there is a directed edge from x to y in $\overline{G}$ if and only if there is a directed edge from x to y in G. (3) For any vertex x in $\overline{G}$, the weight of x in $\overline{G}$ is the total weight of x in G.

- A pseudo-cycle of G is a cycle of the condensed graph of G at some pairwise disjoint blocks. A *pseudo-cycle separator* of G is a pseudo-cycle whose vertices form a separator of G.

- To shorten statements, Algorithm *Join-Paths-and-Cycles* will refer to a class of similar procedures that merge a *constant* number of paths, cycles and vertices into a path or a cycle, using the same idea as in the proof of the Cycle Separator Theorem. Examples of such processes can be found in the overview of the Cycle Separator Algorithm in Subsection 4.1.

- The next algorithm and its slight variations are a key subroutine for processing path-daisy separators and pseudo-cycle separators.

Algorithm 15 (*Find-a-Maximal-Set-of-Paths*)

Input: F, $x_1, ..., x_s$, *and* $y_1, ..., y_t$, *where* (1) F *is any planar digraph such that its boundary is a vertex-simple cycle but not necessarily a directed cycle, and* (2) $x_1, ..., x_s$ *and* $y_1, ..., y_t$ *are the vertices on the boundary and in the clockwise order.*

Output: *a maximal set of pairwise vertex-disjoint paths from* x_i*'s to* y_j*'s through only interior vertices of* F.

Ideas. The following discussion yields an NC algorithm to find an output. Let $\Pi_1 = \{(i,j) \mid$ there is a path from x_i to y_j through interior vertices of $F.\}$; let $\Pi_2 = \{(i,j) \mid$ the index j is the smallest for i such that $(i,j) \in \Pi_1.\}$; further let $\Pi_3 = \{(i,j) \mid$ the index i is the smallest for j such that $(i,j) \in \Pi_2.\}$. Now for each $(i,j) \in \Pi_3$, let $P(i,j)$ be an *arbitrary* path from x_i to y_j through interior vertices of F; let MP be the set of these $P(i,j)$'s. It is straightforward to show that MP is a valid output; furthermore, MP can be computed in parallel as follows. Let m be the number of vertices in F. To obtain the three Π_k's, first compute an $m \times m$ boolean matrix TC such that its $(p,q)^{th}$ entry is 1 if and only if F has a path from the p^{th} vertex to the q^{th} vertex through interior vertices. TC can be computed by slightly modifying the standard doubling-up process for transitive closure; with TC obtained, the Π_k's can be easily computed one by one. To compute an arbitrary $P(i,j)$ for each $(i,j) \in \Pi_3$, first compute from TC the vertex set $V(i,j) = \{x_i, y_j\} \cup \{v \mid v$ is a vertex reachable *both* from x_i *and* to y_i through interior vertices in $F.\}$. These $V(i,j)$'s are pairwise disjoint because of the minimality of the indices i and j; with the disjoint property, the $P(i,j)$'s can be simultaneously computed.

4.3 Trim Blocks off a Pseudo-Cycle Separator

The next algorithm is a detailed description for Step 3 in the overview of the Cycle Separator Algorithm in Subsection 4.1.

Algorithm 16 (*Trim-Blocks-off-a-Pseudo-Cycle-Separator*)

Input: *G and a pseudo-cycle separator C_{in} containing k blocks $B_1, ..., B_k$.*

Output: *a pseudo-cycle separator C_{out} such that C_{out} contains at most one block, and this block is some original B_i.*

Method. The idea is to iteratively cut by half the number of blocks in C_{in}; it takes $O(\log n)$ such cuts to reduce down to one the number of blocks. The cutting process is described as follows.

Step 1. W.l.o.g. assume that C_{in} runs clockwise and $B_1, ..., B_k$ are listed clockwise. Call $B_1, ..., B_{\lceil k/2 \rceil}$ the *upper blocks*; call $B_{\lceil k/2 \rceil + 1}, ..., B_k$ the *lower blocks*. The paths connecting adjacent blocks on C_{in} are called the *external connection paths*. Each block has a vertex at which an external connection path enters (or leaves) the block; call this vertex the *entrance* (or *exit*, respectively) of the block.

Step 2. On each *upper* block, find an arbitrary spanning tree *converging* to the *exit*; in other words, the tree is rooted at the exit and the tree edges point from children to parents. In contrast, on each *lower* block compute an arbitrary spanning tree *diverging* from the *entrance*. Because the blocks are strongly connected, such trees exist; because the blocks are pairwise disjoint, these trees can be computed in parallel. Each block now has a unique tree path between its exit and entrance; call this path the *internal connection path* of the block. Then all internal and external connection paths together form a clockwise cycle; denote this cycle by C_{mid}. If C_{mid} is not yet a cycle separator, w.l.o.g. assume that the interior of C_{mid} is overweight.

Step 3. Let L (or U) be the set of vertices that are on the boundaries of the lower (or upper, respectively) blocks and simultaneously within the interior of C_{mid}. Then use Algorithm Find-a-Maximal-Set-of-Paths in Subsection 4.2 to obtain a maximal set of pairwise vertex-disjoint paths from L to U such that the internal vertices of these paths are within C_{mid} but outside the blocks; call these paths the *division paths*. Then for each division path P from x to y with x in a lower block LB

and y in an upper block UB, the following four paths form a cycle: (1) P, (2) the spanning tree path from y to the exit of UB, (3) the path on C_{mid} from the exit of UB to the entrance of LB, and (4) the spanning tree path from the entrance of LB to y; call such cycles the *division cycles.*

Step 4. The interiors of the division cycles form a increasing chain of the set inclusion. Based on the weights of these interiors, there are three cases: (1) no division cycles have overweight interiors, (2) all division cycles have overweight interiors, or (3) otherwise. All three cases are essentially the same; so only the third case is discussed in detail as follows. Because of the increasing chain property, it is easy to compute in parallel an *adjacent* pair of division paths PL and PR together with their respective division cycles CL and CR such that CL has an overweight interior but CR does not; PL is to the left of PR. The cycles CL and CR divide the interior of C_{mid} into three regions: (1) the left region is outside CL, (2) the central region is inside CL and outside CR, and (3) the right region is inside CR. Because the interior of CL is overweight, the left region cannot be overweight; because the interior of CR is not overweight, the right region is not overweight. Therefore the following three parts form a separator: (1) C_{mid}, (2) the paths PL and PR, and (3) the blocks on the boundary of the central region.

Step 5. Further delete from the above separator some remaining blocks as follows. From the maximality of the division path set, there are no directed paths from L to U within the central region; consequently in the central region, the vertices of either the upper blocks or the lower blocks can be removed from the separator while the remainder of the separator is still a separator. Therefore a new separator can be obtained from (1) C_{mid}, (2) the division cycles CL and CR, and (3) w.l.o.g. the upper blocks on the boundary of the central region; at this point, at most $1/2$ of the B_i's remain. The new separator can be decomposed into two parts: (1) a pseudo-cycle consisting of C_{mid} and the remaining blocks, and (2) two paths which are part of the division cycles CL and CR. Now use Algorithm Join-Paths-and-Cycles in Subsection 4.2 to merge the pseudo-cycle and two paths into

a pseudo-cycle separator; perform this computation on the condensed graph of G at the remaining blocks. The outcome is a pseudo-cycle separator containing at most one half of the original blocks.

4.4 Basics of Path-Daisy Separators

- A *daisy* is a strongly connected planar digraph such that all non-trivial cycles go through the same vertex u. (A trivial cycle is a single vertex or the empty set.) The vertex u is called a *center* of the daisy.

- Let J be a daisy; let e be an edge of J. A *petal bounded by* e in J is a cycle C going through e such that the cycle has a *maximal* interior among all cycles going through e; in other words, for any cycle C' going through e, if the interior of C is a subset of that of C', then $C = C'$.

Lemma 17

For any daisy, there are at most two petals bounded by each edge; furthermore, finding all these petals is in deterministic NC.

Proof. The first half of the lemma. Let J be any daisy; let w be a center of J; let $e = s \rightarrow t$ be any edge of J. Also let T (or S) be the set of edges on paths *from* t *to* w (or respectively, *from* w *to* s). Because J is a daisy centered at w, the boundary of T (or S) *consists* of one or two paths from t to the center (or respectively, from the center to s). Furthermore, the following three items form a cycle: (1) the edge e, (2) either of the two boundary paths of T, and (3) either of the two boundary paths of S. There are at most four such combinations; these combinations include the two petals bounded by e.

The second half of the lemma. The petals can be found by a straightforward implementation of the above discussion, using the planarity algorithm by Klein and Reif [KR86] and a standard transitive closure algorithm. ∎

- A *simple daisy* is one such that the petals bounded by its boundary edges are the only non-trivial cycles in the daisy. A simple daisy has no edges and no vertices inside any petal; moreover, the petals have pairwise disjoint interiors and can be arranged in a circular order around the center of the simple daisy. These properties can be used to visualize a simple daisy; they are also essential for the petal-cutting process in the Subsubsection 4.5.2.

- Let $L = (V, E)$ be a weighted planar digraph; let P be a path of L; let $H = (V', E')$ be a daisy (or simple daisy) such that $V' \subseteq V$ and $E' \subseteq E$. The pair (P, H) is called a *path-daisy separator* (or respectively, *path-simple-daisy separator*) of L if P and H are vertex-disjoint and their vertices together form a separator of L. The edge set of H is not necessarily equal to $E \cap (V' \times V')$; in fact, edges of H are often removed to simplify its structure without turning (P, H) into a non-separator.

4.5 Cut Petals off a Path-Daisy Separator

The next algorithm is a detailed description for Step 2 of the overview of the Cycle Separator Algorithm in Subsection 4.1.

Algorithm 18 (*Convert-a-Path-Daisy-Separator-to-a-Cycle-Searator*)

Input: $\overline{G}$ *and a path-daisy separator* (P_{in}, D_{in}) *of* $\overline{G}$.

Output: *a cycle separator of* $\overline{G}$.

Overview. There are three steps.

Step 1. (Subsubsection 4.5.1.) Convert (P_{in}, D_{in}) to a path-*simple*-daisy separator (P_{mid}, SD_{mid}).

Step 2. (Subsubsection 4.5.2.) Convert (P_{mid}, SD_{mid}) to another path-simple-daisy separator (P_{final}, SD_{final}) such that SD_{final} has at most one petal, or in other words SD_{final} is a cycle.

Step 3. Use Algorithm Join-Paths-and-Daisies in Subsection 4.2 to merge the path P_{final} and the cycle SD_{final} into a cycle separator of $\overline{G}$.

4.5.1 Convert a Path-Daisy Separator to a Path-Simple-Daisy Separator

The next algorithm is a detailed description for Step 1 of Algorithm 18.

Algorithm 19 (*Convert-a-Path-Daisy-Separator-to-a-Path-Simple-Daisy-Separator*)

Input: *a weighted digraph* $\overline{G}$ *and a path-daisy separator* (P_{in}, H_{in}).

Output: *a path-simple-daisy separator* (P_{out}, H_{out}) *and the petals of* H_{out}.

Method. First find all petals of H_{in}; the planar embedding for H_{in} is part of that for $\overline{G}$. Then there are two cases based on whether any petal has an overweight interior. Case I: No petal has an overweight interior; Case II: At least one petal has an overweight interior. These two cases are discussed separately as follows.

Case I. No petal has an overweight interior. Let L be the boundary of H_{in}; a *boundary sink* (or *source*) is a vertex on L with zero out-degree (or in-degree, respectively) on L. Because H_{in} is a daisy, for any boundary sink (source), H_{in} has a path from the sink to the center of H_{in} (or respectively, from the center to the source) such that the path and L are disjoint except at its end(s); call such a path a *petal path.* Compute in parallel a set of petal paths, one for each source and one for each sink; because H_{in} is a daisy, the petals in any such set are pairwise disjoint except that they all share the center. These chosen petals divide the interior of H_{in} into disjoint regions; each region is bounded by the petal paths of an adjacent source-sink pair; so the boundary of each region is a cycle. Delete from H_{in} all edges and all vertices within each region; then P_{in} and the remainder of H_{in} still form a path-daisy separator of $\overline{G}$ because every cycle is contained in a petal and because no petal of H_{in} has an overweight interior. After the deletion, the remainder of H_{in} is a simple daisy and the boundaries of the disjoint regions are its new petals.

Case II. At least one petal has an overweight interior. To reduce this case to Case I, H_{in} can be shrunk by the following deletion analysis. Take a pair of petals A and B with overweight interiors

such that A and B run in the same direction, say, clockwise. Because both A and B have overweight interiors, the following three areas cannot be overweight: (1) the area outside both A and B, (2) the area inside A but outside B, and (3) the area outside A but inside B. In other words, only the area inside both A and B may be overweight; call this area the *central area*. Based on this possibility, there are two cases. Case A: The central area is not overweight. Then the two cycles already form a separator; keep these two cycles. Case B: The central region is overweight. Then the boundary of this central area is a cycle C because H_{in} is a daisy and A and B run in the same direction; replace A and B by C.

Now pair up the petals with overweight interiors such that each pair runs in the same direction; perform the above analysis in parallel for all pairs. The outcome is either at least one pair is in Case A or all pairs are in Case B. In the former outcome, save only one pair; in the latter outcome, there remain one half as many overweight cycles as before the analysis. By repeating the analysis $O(\log n)$ times, at most two cycles left; the following three cases and their symmetries are the only possibilities: (1) one clockwise cycle from Case B, (2) two clockwise cycles from Case A, or (3) one clockwise cycle and one counter-clockwise cycle both from Case B. The three cases are discussed separately as follows.

Case II-(1). Let R be the remaining clockwise cycle; let H_{mid} be the H_{in} without the area outside R; then (P_{in}, H_{mid}) is still a separator. Because of the above shrinking process, the interior of R is the intersection of all petals of H_{in} with overweight interiors; this fact can be used to show that no petal of H_{mid} except R has an overweight interior. Therefore, a segment CH can be chopped off R such that the remainder of H_{mid} is still a daisy; however, because the loss of CH destroys R, H_{mid} now has no petal of an overweight interior. To reduce the case back to Case I, use Algorithm Join-Paths-and-Cycles in Subsection 4.2 to merge CH and P_{in} into a path.

Case II-(2). Let $R1$ and $R2$ be the two remaining cycles; then $R1$ and the part of $R2$ within $R1$ form a separator. The remainder of $R2$ are actually chords on $R1$; to get a simpler separator,

start with $R1$ and add the chords one by one. Adding a chord in effect chops off a region and forms a smaller cycle; in other words, the chords induce a decreasing chain of cycles. In this chain some pair of adjacent cycles still form a separator; finding such a pair is in NC. Moreover, an adjacent pair is just a cycle and a chord; so use Algorithm Join-Paths-and-Cycles in Subsection 4.2 to merge the adjacent pair into a cycle separator. This is a better output than requested for the output.

Case II-(3). Let CR and CL be the remaining cycles. Delete from H_{in} everything outside CR; then the remainder and P_{in} still form a path-daisy separator. Moreover, the remainder of CL consists of chords on CR; these chords divide the interior of CR into disjoint regions. Because the regions are disjoint, either exactly one region is overweight or no region is overweight. In the former case, delete everything in all the regions except the overweight region; the case is reduced to Case II-(1). In the latter case, delete everything in all the regions; the case is reduced to one very close to Case II-(2).

4.5.2 Convert a Path-Simple-Daisy Separator to a Path-Cycle Separator

The next algorithm is a detailed description for Step 2 of Algorithm 18.

Algorithm 20 (*Convert-a-Path-Simple-Daisy-Separator-to-a-Path-Cycle-Separator*)

Input: *a weighted digraph* $\overline{G}$, *a path-simple-daisy separator* (P_{in}, H_{in}) *of* $\overline{G}$, *and the petals* $E_1, ..., E_k$ *of* H_{in}.

Output: *a path-simple-daisy separator* (P_{out}, H_{out}) *such that* H_{out} *has at most one petal, or in other words* H_{out} *is a cycle of* $\overline{G}$.

Method. The idea is to repeatedly cut by half the number of petals in H_{in} while building up P_{in}; it takes $O(\log n)$ such cuts to reduce to one the number of petals. The petal cutting process is described as follows.

Step 1. W.l.o.g. let $E_1, ..., E_k$ be listed in the same circular order as their edges appear on the boundary of H_{in}. Call $E_1, ..., E_{\lceil k/2 \rceil}$ the *upper petals*; call $E_{\lceil k/2 \rceil + 1}, ..., E_k$ the *lower petals*. Let U

(or L) be the set of vertices on the boundary of H_{in} and simultaneously on the upper (or lower, respectively) petals. U and L contain one or two common vertices; delete them from U and L for simplicity.

Step 2. Use Algorithm Find-a-Maximal-Set-of-Paths in Subsection 4.2 to compute a maximal set of pairwise disjoint paths from L to U such that the internal vertices of these paths are outside the boundary of H_{in} and not on P_{in}; call such paths the *division paths.* For each division path, there is a cycle consisting of the path and parts of a lower petal and an upper one; the cycle goes through the center of H_{in}; call such cycles the *division cycles.* Because H_{in} is simple, the division cycle is unique, with one minor exception, for any division path. The exception happens when a division path ends at the shared vertex of two adjacent petals; in such a case, use some simple resolution rule such that the interiors of any two division cycles either are disjoint or contain one another.

Step 3. These division cycles form two groups, counter-clockwise and clockwise. If two cycles are in the opposite directions, their interiors are disjoint; for the cycles in the same group, their interiors form an increasing chain of the set inclusion. There are several cases based on whether any division cycle has an overweight interior. All of them are essentially the same; so the following discussion focuses on the case that in the clockwise group at least one cycle has an overweight interior and at least one cycle does not have an overweight interior. From the increasing chain property, there is an adjacent pair of clockwise division cycles $C1$ and $C2$ such that the larger $C1$ has an overweight interior and the smaller $C2$ does not; finding such an adjacent pair is in NC.

Step 4. Now discard petals as follows. Call the region between $C1$ and $C2$ the *central region.* Because $C1$ does not have an overweight *exterior* and because $C2$ does not have an overweight *interior*, the following three items form a separator of $\overline{G}$: (1) P_{in}, (2) C_1 and C_2, and (3) the petals in the central region. The third item can be further discarded as follows. From the maximality of the division path set, there are no directed paths from L to U through the central region; this implies

that any strongly connected component of the central region cannot have vertices from L and U at the same time. Therefore, w.l.o.g. the vertices of L in the the central region can be deleted without turning the 3-item separator into a non-separator; at this point, no lower petals survive the cut even though $C1$ and $C2$ contain segments of them. The remainder of the three-item separator has another decomposition: (1) the simple daisy consisting of the remaining petals, (2) two paths which are part of $C1$ and $C2$, and (3) the path P_{in}. The first item is a simple daisy of at most $\lceil k/2 \rceil$ petals; the other two items can be merged into a path by Algorithm Join-Paths-and-Cycles in Subsection 4.2. The resulting path-simple-daisy separator is a desired output.

5 Conclusions

This paper has given very simple proofs to two rather unexpected theorems, namely, the Path Separator Theorem and the Cycle Separator Theorem. The proofs are based on depth-first search, and have many algorithmic implications. In sequential computation, the proofs have been translated into very efficient algorithms for computing path and cycle separators of any graphs. In parallel computation, finding path separators, finding cycle separators, and conducting depth-first search for general digraphs have been shown to be deterministically NC-equivalent; the proof combines the ideas in the two separator theorems with a general technique of using cycle separators to conduct parallel depth-first search for general digraphs.

The paper has also presented a deterministic NC algorithm for computing a cycle separator for any weighted planar digraph; consequently, finding a depth-first search forest in any planar digraph is also in deterministic NC. The cycle separator algorithm runs in $O(\log^5 n)$ time and $O(n^4)$ processors on an EREW PRAM; the corresponding depth-first search algorithm runs in $O(\log^7 n)$ time and $O(n^4)$ processors. Such complexity is beyond practical application; it remains an open question to find algorithms with much better complexity.

References

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

[And85] Richard Anderson. A parallel algorithm for the maximal path problem. In *ACM Symposium on Theory of Computing*, pages 33–37, 1985.

[FJ86] Greg N. Frederickson and Ravi Janardan. Separator-based strategies for efficient message routing. In *IEEE Symposium on Foundations of Computer Science*, pages 428–437, 1986.

[GB84] Ratan K. Ghosh and G. P. Bhattacharjee. A parallel search algorithm for directed acyclic graphs. *BIT*, 24:134–150, 1984.

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

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

[JV83] Donald B. Johnson and Shankar M. Venkatesan. Partition on planar flow networks. In *IEEE Symposium on Foundations of Computer Science*, pages 259–264, 1983.

[KR86] Philip N. Klein and John H. Reif. An efficient parallel algorithm for planarity. In *IEEE Symposium on Foundations of Computer Science*, pages 465–477, 1986.

[KRS85] Clyde P. Kruskal, Larry Rudolph, and Marc Snir. The power of parallel prefix. *IEEE Transactions on Computers*, c-34(10):965–968, October 1985.

[Lei80] Charles E. Leiserson. Area-efficient graph layouts (for VLSI). In *IEEE Symposium on Foundations of Computer Science*, pages 270–281, 1980.

[Lov85] L. Lovász. Computing ears and branchings. In *IEEE Symposium on Foundations of Computer Science*, pages 464–467, 1985.

[LRT79] R.J. Lipton, D.J. Rose, and R.E. Tarjan. Generalized nested dissection. *SIAM Journal of Numerical Analysis*, 16:346–358, 1979.

[LT79] R.J. Lipton and R.E. Tarjan. A separator theorem for planar graphs. *SIAM Journal of Applied Mathematics*, 36:177–189, 1979.

[Mil84] Gary L. Miller. Finding small simple cycle separators for 2-connected planar graphs. In *ACM Symposium on Theory of Computing*, pages 376–382, 1984.

[PR87] Victor Pan and John Reif. *Fast and Efficient Solution of Path Algebra Problems.* Technical Report 3, Computer Science Department, State University of New York at Albany, 1987.

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

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

[SV85] Catherine A. Schevon and Jeffrey Scott Vitter. *A Parallel Algorithm for Recognizing Unordered Depth-First Search.* Technical Report 21, Department of Computer Science, Brown University, 1985.

[Val81] L.G. Valiant. Universality considerations in VLSI circuits. *IEEE Transactions on Computers*, 30(2):135–140, February 1981.