AD-A056 888 ILLINOIS UNIV AT URBANA-CHAMPAIGN COORDINATED SCIENCE LAB F/G 9/2
PARALLEL ALGORITHMS FOR GRAPH THEORETIC PROBLEMS.(U)
AUG 77 C D SAVAGE DAAB07-72-C-0259
UNCLASSIFIED R-784 NL

1 OF 2
AD
A056888

LEVEL II

REPORT ACT-4 AUGUST, 1977

COORDINATED SCIENCE LABORATORY

APPLIED COMPUTATION THEORY GROUP

AD A056888

PARALLEL ALGORITHMS FOR GRAPH THEORETIC PROBLEMS

CARLA DIANE SAVAGE

AD No.
DDC FILE COPY

DDC
RECEIVED
AUG 2 1978
D

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

REPORT R-784 UILU-ENG 77-2231

UNIVERSITY OF ILLINOIS - URBANA, ILLINOIS

78 07 10 037

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) PARALLEL ALGORITHMS FOR GRAPH THEORETIC PROBLEMS		5. TYPE OF REPORT & PERIOD COVERED Technical Report
		6. PERFORMING ORG. REPORT NUMBER R-784; UILU-ENG 77-2231
7. AUTHOR(s) Carla Diane Savage		8. CONTRACT OR GRANT NUMBER(s) NSF MCS 76-17321 DAAB-07-72-C-0259
9. PERFORMING ORGANIZATION NAME AND ADDRESS Coordinated Science Laboratory University of Illinois at Urbana-Champaign Urbana, Illinois 61801		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Joint Services Electronics Program		12. REPORT DATE August 1977
		13. NUMBER OF PAGES 126
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Computational Complexity
Parallel Computation
Graph Algorithms
Minimum Spanning Trees
Cycles, Bridges, Dominators

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The existence of parallel computers has motivated the development of parallel problems solving techniques for many problems. We study techniques for solving graph problems on an unbounded parallel model of computation. It is shown that solutions to graph problems can be organized to reveal a large amount of parallelism, which can be exploited to substantially reduce the computation time. Precisely, for an appropriate measure of time complexity, algorithms of time complexity $O(\log^2 n)$ are developed to solve each of the following problems for graphs with n vertices: finding minimum spanning trees, biconnected components,

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

78 07 10 038

20. ABSTRACT (continued)

dominators, bridges, cycles, cycle bases, and shortest cycles. The number of processors needed to execute each algorithm is bounded above by a polynomial function of n. It is shown that 2 logn + c is a lower bound on thetime required to solve each of these graph problems. Thus, the algorithms obtained have time complexities which are optimal to within a factorof logn.

LEVEL II

UILU-ENG 77-2231

PARALLEL ALGORITHMS FOR GRAPH THEORETIC PROBLEMS

by

Carla Diane Savage

This work was supported by the National Science Foundation under Grant NSF MCS 76-17321; additional supported was provided by the Joint Services Electronics Program (U.S. Army, U.S. Navy and U.S. Air Force) under Contract DAAB-07-72-C-0259.

Reproduction in whole or in part is permitted for any purpose of the United States Government.

Approved for public release. Distribution unlimited.

ACCESSION for
NTIS White Section
DDC Buff Section
UNANNOUNCED
JUSTIFICATION
BY
DISTRIBUTION/AVAILABILITY CODES
Dist. AVAIL. and/or SPECIAL
A

DDC
RECEIVED
AUG 2 1978
D

PARALLEL ALGORITHMS FOR GRAPH THEORETIC PROBLEMS

BY

CARLA DIANE SAVAGE

B.S., Case Western Reserve University, 1973
M.S., University of Illinois, 1975

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Mathematics
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1977

Thesis Adviser: Professor David E. Muller

Urbana, Illinois

PARALLEL SOLUTIONS TO GRAPH THEORETIC PROBLEMS

Carla Diane Savage, Ph.D.
Coordinated Science Laboratory and
Department of Mathematics
University of Illinois at Urbana-Champaign, 1977

The existence of parallel computers has motivated the development of parallel problem solving techniques for many problems. We study techniques for solving graph problems on an unbounded parallel model of computation. It is shown that solutions to graph problems can be organized to reveal a large amount of parallelism, which can be exploited to substantially reduce the computation time. Precisely, for an appropriate measure of time complexity, algorithms of time complexity $O(\log^2 n)$ are developed to solve each of the following problems for graphs with n vertices: finding minimum spanning trees, biconnected components, dominators, bridges, cycles, cycle bases, and shortest cycles. The number of processors needed to execute each algorithm is bounded above by a polynomial function of n. It is shown that 2 logn + c is a lower bound on the time required to solve each of these graph problems. Thus, the algorithms obtained have time complexities which are optimal to within a factor of logn.

ACKNOWLEDGEMENT

I would like to express my gratitude and appreciation to my advisor, David Muller, for his advice and support. In addition, I thank the National Science Foundation for its financial assistance during the past year.

TABLE OF CONTENTS

Chapter		Page
1	INTRODUCTION	1
	1.1 Models of Computation	1
	1.2 Complexity Measures	3
	1.3 Previous Results	6
	1.4 Outline of Thesis	8
2	DEFINITIONS AND BASIC ALGORITHMS	11
	2.1 Definitions	11
	2.2 Basic Algorithms	14
3	SPANNING TREES	22
	3.1 Definitions	22
	3.2 The Minimum Spanning Tree Algorithm	23
	3.3 Applications of the MST Algorithm	46
	3.4 Directed Spanning Trees	50
4	BICONNECTED COMPONENTS	55
5	DOMINATORS	67
6	BRIDGES	72
	6.1 First Bridge Algorithm	72
	6.2 Preliminary Algorithms	77
	6.3 Improved Bridge Algorithm	85
	6.4 Bridge Connected Components	92
7	CYCLES	95
	7.1 Cycles in an Undirected Graph	95
	7.2 Shortest Cycles	102

Chapter | Page

8 LOWER BOUNDS. 110

8.1 A Fan-In Theorem 110
8.2 The Aanderaa-Rosenberg Theorem 114
8.3 Lower Bound Results 115

9 CONCLUSION 121

REFERENCES 123

VITA 126

Chapter 1

INTRODUCTION

Computers now exist which are capable of performing several independent operations simultaneously [3,7,10,24,25,31]. This fact is the motivation for much recent research into problem solving methods which take advantage of this parallelism to minimize computation time. Such methods have been developed not only for problems like vector addition, which is inherently parallel, but also for problems such as solving a system of linear equations [8], evaluating an arithmetic expression [16], and sorting an array [18]. Most recently, graph problems have been considered and algorithms for finding the connected components of a graph have been developed [2,11]. This paper is a study of parallelism in other graph theoretic problems. The problems considered include finding, for a given graph, a minimum spanning tree, a cycle basis, the biconnected components, the bridges, and the dominators. For each problem, a parallel algorithm is developed and analyzed. In addition, theoretical lower bounds on the time required to solve the problems are found. The model of computation which is used is the unbounded parallel model discussed in Section 1.1.

1.1 Models of Computation

A meaningful study of the development and analysis of algorithms in which several operations may be performed simultaneously requires a precise model of computation. Existing parallel machines

differ widely in their characteristics and the study of their operation is complicated by hardware considerations. Although it is of practical interest to develop algorithms to solve problems on a particular computer, the structure and performance of these algorithms will depend not only on the problem at hand, but on the advantages and limitations of the computer as well. A more fundamental approach to parallel computing is to study the structure of problem solutions and seek ways to reorganize solutions so that operations can be performed simultaneously to reduce the computation time. Two commonly used models of computation, which are suitable for this approach, are discussed below.

Let B be a set of binary operations, which include binary comparisons. The k-parallel model consists of k identical, independently controlled processors, each of which is capable of performing operations in B. Each processor has access to an arbitrarily large common memory. The k-parallel model operates synchronously, in a sequence of steps, called time units. Initially, the input is assumed to be stored in the memory. During any time unit, each processor may take its operands from the memory, perform an operation in B, and store the result of the operation in the memory. During no time unit may two processors store a result in the same memory location, although they may read the content of the same memory location.

The unbounded parallel model is the same as the k-parallel model, except that the number of processors is unbounded. The unbounded parallel model of computation is the one which will be used in this paper. Notice that this model is sufficiently general to handle adaptive

operations. For example, let b be a function of n variables, x_1, . . ., x_n, where $b(x_1, . . ., x_n) \varepsilon \{0,1\}$. Assume that b can be computed on the unbounded parallel model, given $(x_1, . . ., x_n)$. Let $g(y_1, . . ., y_m)$ and $h(z_1, . . ., z_k)$ be arbitrary functions which can be computed on the unbounded parallel model, given $(y_1, . . ., y_m)$ and $(z_1, . . ., z_k)$, respectively. For given values of

$$x_1, . . ., x_n, y_1, . . ., y_m, z_1, . . ., z_k,$$

let f be a function which has the value $g(y_1, . . ., y_m)$ if $b(x_1, . . ., x_n) = 0$ and $h(z_1, . . ., z_k)$ if $b(x_1, . . .,x_n) = 1$. Then

$$f(x_1, . . ., x_n, y_1, . . ., y_m, z_1, . . ., z_k)$$

$$= (1-b(x_1, . . ., x_n))\, g(y_1, . . ., y_m)$$

$$+ b(x_1, . . ., x_n)h(z_1, . . ., z_m).$$

Thus, f can be computed on the unbounded parallel model, under the assumption that the set B contains the operations of addition, subtraction, and multiplication.

1.2 Complexity Measures

In order to evaluate the performance of an algorithm developed to solve a problem on a particular model of computation, parameters are needed to measure its efficiency. On the two models above, meaningful measures of the complexity of an algorithm are the number of time units used and the amount of storage required to perform the

algorithm. Another measure, on the unbounded parallel model, is the maximum number of processors actually used to execute a given algorithm. The time, storage, and number of processors used by an algorithm which solves a certain class of problems may be less for "small" problems in the class than for "large" ones. It is thus important to have a measure of the "size" of a problem and to determine the time, storage, and number of processors used by an algorithm to solve a given problem as a function of the size of the problem.

Let C be a class of problems for which some appropriate measure of size has been determined. (For example, the measure of the size of a problem for a graph could be the number of vertices of the graph. The size of a polynomial evaluation problem could be the degree of the polynomial.) For $p \in C$, let $|p|$ be the size of p. Let A be an algorithm which solves problems in C on a k-parallel or unbounded parallel model. For $p \in C$, let $t(p)$ be the number of time units, $s(p)$, the number of memory locations, and $q(p)$, the number of processors used to solve problem p using algorithm A. Then the time complexity of the algorithm A is a function T_A, defined on the set $X = \{m \mid m \text{ is a non-negative integer and for some } p \in C, |p| = m\}$, where for $n \in X$,

$$T_A(n) = \max \{t(p) \mid p \in C \text{ and } |p| = n\}.$$

The storage complexity of A is the function S_A, where

$$S_A(n) = \max \{s(p) \mid p \in C \text{ and } |p| = n\}$$

and the processor complexity of A is the function Q_A, where

$$Q_A(n) = \max \{q(p) \mid p \in C \text{ and } |p| = n\}.$$

The time, storage, and processor complexities provide ways to evaluate and compare algorithms developed for implementation on the k-parallel or unbounded parallel model. It is also of interest to compare the complexity of a given parallel algorithm to the complexity of a sequential algorithm which solves the same problem. Sequential algorithms have been studied extensively and the most commonly used model of computation is a random access machine (RAM) [1]. A RAM is similar to a 1-parallel processor where input is read from an input tape, rather than from the memory. Any algorithm designed to be executed on a RAM in time T can be modified to work on the 1-parallel model in time at most T.

In developing efficient algorithms on a k-parallel model, the primary goal is to find, for a given problem, an algorithm of minimum time complexity, using the k processors available. Let A be a k-parallel algorithm with time complexity T(n). During execution of A, at most k binary operations are performed in any time unit. Modifying A so that these operations are performed sequentially, we obtain a sequential algorithm, A', which solves the same problem as A and has time complexity at most $k \cdot T(n)$. Thus, if the best serial algorithm for a given problem has time complexity f(n), the goal is to find a k-parallel algorithm of time complexity which is bounded above by $c \cdot f(n)/k$, for some constant c. When a "fast" algorithm is found, it is desirable to increase its efficiency by minimizing the storage complexity.

When an unbounded parallel model is used as the model of computation, again the goal is to find, for a given problem, an algorithm of minimal time complexity, regardless of how many processors must be used. When a "fast" algorithm has been found, it is of practical interest to try to minimize the number of processors used without increasing the time complexity by more than a constant factor.

Before we mention some results which have been obtained using the unbounded parallel model of computation, the "O" notation should be defined. Let f and g be nonnegative real valued functions defined on the set of nonnegative integers. Then $g(n) = O(f(n))$, read "g(n) is of order f(n)," if, for some constant c, $g(n) \leq cf(n)$ for all but finitely many values of n [1]. Expressions like "algorithm A takes time O(f(n))" will be used throughout this paper to mean "the time complexity of algorithm A is O(f(n))."

1.3 Previous Results

To show the speedup in computation time which can result by solving a problem in parallel, and to see the number of processors needed to achieve such a speedup, some results which have been obtained are displayed in Table 1. For each problem listed, the table gives the time and processor complexities of the best known parallel algorithm which solves the problem on the unbounded parallel model. For comparison, the time complexity of the fastest known sequential algorithm for each problem is given.

Table 1

Problem	Parallel Time Complexity		Parallel Processor Complexity		Sequential Time Complexity	
Evaluating a polynomial of degree n	$\log n + O((\log n)^{1/2})$[1]		n	[17]	$2n$	[5]
Evaluating an arithmetic expression with n variables (each occurring once) involving only addition, multiplication, and division	$2.88 \log n + 1$		$O(n^{1.44})$	[16]	$n-1$	
Multiplying two n x n matrices	$O(\log n)$		$O(n^{\log 7})$	[6]	$O(n^{\log 7})$	[27]
Inverting an n x n matrix			$O(n^4)$	[8]		
Solving a system of n linear equations	$O(\log^2 n)$	[8]	$O(n^{3.31}/\log^2 n)$	[19]	$O(n^{\log 7})$	[5]
Sorting a list of n elements	$O(\log n)$		$n\lceil \log n\rceil$	[18]	$n\log n$	[1]
Finding the connected components of an undirected graph with n vertices and m edges	$O(\log^2 n)$		n^2	[11]	$O(n+m)$	[21]
Finding the strongly connected components of a directed graph with n vertices and m edges	$O(\log^2 n)$		n^3	[2]	$O(n+m)$	[28]

[1]All logarithms are to the base two.

1.4 Outline of Thesis

In this paper, parallel algorithms are presented for several graph problems, based on the unbounded parallel model of computation presented in Section 1.1. Although parallelism in graph problems has only recently been considered, sequential algorithms for graph problems have been studied extensively and efficient algorithms exist for many problems, including finding the biconnected components of a graph [1], finding a minimum spanning tree of a graph [32], and deciding whether a graph is planar [14]. Unfortunately, one of the major tools used in efficient sequential algorithms, the depth first search of Tarjan [28], cannot be adapted to yield efficient algorithms on a model with unbounded parallelism. On the other hand, some inefficient serial algorithms can be transformed into fast parallel algorithms. The algorithms presented in this paper have been developed by combining sequential techniques with new techniques, designed to exploit parallelism.

The graph problems which will be considered here are the following. For a given connected, undirected graph G, find a spanning tree of G, a cycle of G, a cycle basis for G, the bridges and bridge connected components of G, and the biconnected components of G. For a given connected, undirected, weighted graph G, find a minimum spanning tree of G. For a given connected, directed graph G, find a cycle of G, a shortest cycle of G, and the dominators and dominator tree of G. Using the adjacency matrix of the graph as input (or the weight matrix in the case of a weighted graph), we develop parallel algorithms of

time complexity $O(\log^2 n)$ to solve each of these problems, where n is the number of vertices of the graph. It is shown that for each algorithm, there is a polynomial in n which is an upper bound on the number of processors required to execute the algorithm on a graph with n vertices.

Lower bounds for graph problems are discussed in Chapter 8. It is shown for many problems that any algorithm which solves the problem on the unbounded parallel model, using the adjacency or weight matrix as input, requires time at least $2 \log n + c$, for some constant c.

It is of interest to compare the time and processor complexities of the parallel algorithms presented here with the time complexity of the corresponding sequential algorithm. Let $T(n)$ and $P(n)$ be the time and processor complexities, respectively, of a parallel algorithm, A, for a certain problem. Let $T_1(n)$ be the time complexity of the fastest sequential algorithm, A', which solves the same problem. The ratio $S(n) = T_1(n)/T(n)$, called the <u>speedup</u>, is a measure of the decrease in computation time which results by using algorithm A instead of algorithm A'. To compare the overall efficiency of algorithms A and A', note that A' performs at most $T_1(n)$ operations, while A performs at most $T(n) \cdot P(n)$ operations. The ratio $E(n) = S(n)/P(n)$, called the <u>efficiency</u>, is a comparison of the total work done by both algorithms. A small function $E(n)$ (for example, $E(n) \leq 1/n^2$) indicates that a large amount of waste is incurred by using $P(n)$ processors to solve the given problem.

Each graph problem considered here can be solved by a sequential algorithm with time complexity bounded above by a polynomial function of the number of vertices of the graph. It has been shown for sequential algorithms that different representations of a problem can yield algorithms of different time complexities. (For example, it can be shown that if a graph G, with n vertices, is represented by its adjacency matrix, at least cn^2 time units are required to determine whether or not G is planar [22]. On the other hand, Hopcroft and Tarjan [14] have developed an algorithm which determines planarity in time $O(n)$ using an adjacency list representation [1] of G.) Since all algorithms presented here use an adjacency or weight matrix representation of a graph, it is most appropriate to compare them to sequential algorithms which use the same representation. But it is also of general interest to compare the parallel algorithms to the fastest known serial algorithms. Thus, whenever possible, the time complexities of the fastest known sequential algorithms in both cases will be given, for comparison with the corresponding parallel algorithm.

Definitions, results, and basic algorithms which are used in this paper are discussed in Chapter 2.

Chapter 2

DEFINITIONS AND BASIC ALGORITHMS

2.1 Definitions

An undirected graph, $G = (V,E)$, consists of a finite, non-empty set V of vertices and a set E of unordered pairs, (x,y), of elements of V, called edges. If $(x,y) \varepsilon E$, x and y are adjacent and (x,y) is incident with x and y. G is simple if $(x,x) \notin E$ for all $x \varepsilon V$. It will be assumed that all undirected graphs in this paper are simple. A graph $H = (V_H,E_H)$ is a subgraph of G, written $H \subseteq G$, if $V_H \subseteq V$ and $E_H \subseteq E$. A path in G is a subgraph $P = (V_P,E_P)$ of G where $V_P = \{x_1, \ldots, x_k\}$ and $E_P = \{(x_i,x_{i+1}) \mid i = 1, \ldots, k-1\}$. The path P joins x_1 and x_k and the length of P is $k - 1$. We will sometimes represent P as a sequence $x_1, \ldots, x_k$. The path P is simple if $x_i \neq x_j$ for $i,j \varepsilon \{1, \ldots, k\}$; P is a cycle if $x_1 = x_k$, $k \geq 4$, and $x_1, \ldots, x_{k-1}$ is a simple path. The union of two graphs, $G = (V,E)$ and $G' = (V',E')$, is the graph $G \cup G' = (V \cup V', E \cup E')$. In this paper, the vertices of a graph, $G = (V,E)$, with n vertices will usually be identified with the integers 1 through n, that is, $V = \{1, \ldots, n\}$. The adjacency matrix of G is an $n \times n$ matrix A, where for $i, j \varepsilon V$,

$$A(i,j) = \begin{cases} 1, & \text{if } (i,j) \varepsilon E \\ 0, & \text{otherwise.} \end{cases}$$

A directed graph, $G = (V,E)$, consists of a finite, nonempty

set V of vertices and a set E of ordered pairs, $\overrightarrow{(x,y)}$, of elements of V, called edges. If $\overrightarrow{(x,y)} \in E$, then x is the tail of $\overrightarrow{(x,y)}$, y is the head of $\overrightarrow{(x,y)}$, and $\overrightarrow{(x,y)}$ leaves x and enters y. The graph G is simple if $(x,x) \notin E$ for all $x \in V$. Subgraphs and union of graphs are defined for directed graphs as for undirected graphs. A path P in G is a subgraph $P = (V_P,E_P)$ of G where $V_P = \{x_1, \ldots, x_k\}$ and $E_P = \{\overrightarrow{(x_i,x_{i+1})} \mid i = 1, \ldots, k - 1\}$; P is a path from x_1 to x_k and the length of P is k - 1. We will sometimes represent P as a sequence $x_1 \to x_2 \to \ldots \to x_k$. The path P is simple if $x_i \neq x_j$ for $i, j \in \{1, \ldots, k\}$; P is a cycle if $x_1 = x_k$ and the path $x_1 \to x_2 \to \ldots \to x_{k-1}$ is a simple path. If G = (V,E), where V = {1, . . ., n}, the adjacency matrix of G is the n x n matrix A, where for $i, j \in V$

$$A(i,j) = \begin{cases} 1, & \text{if } \overrightarrow{(i,j)} \in E \\ 0, & \text{otherwise.} \end{cases}$$

An undirected (directed) graph G = (V,E) is connected (strongly connected) if for all $i, j \in V$ there is a path in G joining i and j (from i to j). A connected component (strongly connected component) of G is a maximal connected (strongly connected) subgraph of G.

A forest is an undirected graph which contains no cycles. A connected forest is a tree. A rooted tree, (T,r), is a tree T = (V,E) with a distinguished vertex $r \in V$. If $i \in V$ and $i \neq r$, the father of i in (T,r) is the vertex $j \in V$ which is adjacent to i on the shortest path in T joining i and r. Then i is the son of j. If j lies on the

shortest path in T joining i and r, then j is an *ancestor* of i and i is a descendant of j. If $i \neq j$, j is a *proper ancestor* of i and i is a *proper descendant* of j. In this paper, a *directed rooted tree*, (T,r), is defined to be a directed graph $T = (V,E)$ with a distinguished vertex $r \in V$, in which there is no edge of E leaving r and for each $i \in V$ with $i \neq r$, there is a unique path in T from i to r.[2] If $\overrightarrow{(i,j)} \in E$, j is the *father* of i and i is the *son* of j. If there is a path in T from i to j then i is a *descendant* of j (*proper descendant* if $i \neq j$) and j is an *ancestor* of i (*proper ancestor* if $i \neq j$). A *directed forest* is a union of directed rooted trees with pairwise disjoint vertex sets. Let (T,r) be a rooted tree or a directed rooted tree, where $T = (V,E)$. For i, $j \in V$, the *youngest common ancestor* of i and j in T is the vertex $k \in V$ such that k is an ancestor of i and j and if $v \in V$ is an ancestor of i and j, then v is an ancestor of k.

Let R be a binary relation on a set S. Then R is a *total order* (or linear order) of S if R is reflexive, antisymmetric, and transitive and for all x, $y \in S$, either $x\ R\ y$ or $y\ R\ x$. The *minimum* of S with respect to R, written $\min_R S$, is the element $x \in S$ such that $x\ R\ y$ for all $y \in S$. The *maximum* of S, $\max_R S$, is defined analogously.

Let $\mathbb{R}$ be the set of real numbers and Z, the integers. The *ceiling function* is a function $f : \mathbb{R} \to Z$, where

$$f(x) \text{ (written } \lceil x \rceil) = \min \{z \in Z \mid x \leq z\}.$$

---

[2]Tarjan calls this an anti-tree. In his definition of directed rooted tree, the direction of each edge is reversed.

The floor function is a function $g : \mathbb{R} \to Z$, where

$$g(x) \text{ (written } \lfloor x \rfloor) = \max \{z \in Z \mid z \leq x\}.$$

Inequalities involving logarithms and the ceiling function are used throughout this paper. The following results will be used casually, when they are needed.

1. There is a constant c such that
   $\lceil x \rceil \leq cx$ for all $x \geq 1$
2. Let k be any number. Then there is a constant c such that $x + k \leq cx$ for all $x \geq 1$.

Other definitions will be made throughout the paper as the need arises.

2.2 Basic Algorithms

In this paper, parallel algorithms are sought for various graph problems. The strategy used to develop an algorithm for a given problem is to devise a technique which reduces the solution of the problem to the solution of a sequence of problems for which efficient parallel algorithms are known. In anticipation of such later use, some problems for which parallel algorithms have been developed are discussed below.

Finding the Minimum of n Numbers

Let S be a set of n elements and let R be a total order of the set S. Assume that the relation R is such that each processor of

the unbounded parallel model can calculate $\min_R \{x,y\}$, for $x, y \in S$, in c time units, for some constant c. (Although R will most often refer to the usual ordering on the integers, in some cases S will represent a set of vertices of a graph and R will be a relation on vertices.) For simplicity, assume n is a power of two. The minimum element of S can be found by setting up a tournament in the following way. In the first round, the elements of S are paired off and the minimum element of each pair is called the winner. In the second round, these winners are paired off for new matches and the winners of these matches are determined as before. The process continues until there is only one element which has been a winner after every round. This element is the minimum element of S. Since $n/2^t$ winners remain after round t, log n matches must be carried out before a single winner is obtained.

ALGORITHM MIN (1) below describes how this procedure is implemented on the unbounded parallel model of computation described in Section 1.1.

ALGORITHM MIN (1)

Assume that n elements are stored in a 1 x n array A_0.

Find $\min \{A_0(1), \ldots, A_0(n)\}$ in the following way.

1. Let t = 0.
2. If t = log n, stop the computation, having found $A_t(1)$, the minimum of $A_0(1), \ldots, A_0(n)$. Otherwise, increment t by 1.
3. Partition $\{A_{t-1}(1), \ldots, A_{t-1}(n/2^{t-1})\}$ into $n/2^t$ sets, $S_t^{(1)}$,

. . ., $S_t^{(n/2^t)}$, of two elements each. Assign one processor to each set $S_t^{(i)}$ to compute $A_t(i)$, a minimum of the two elements of $S_t^{(i)}$. Then return to step 2.

The time and processor requirements for each step are as follows. In step 1, one processor is assigned to store a number in the memory. This takes at most 1 time unit. In step 2, one processor is assigned to test whether or not $t = \log n$. Depending on the result, the processor may stop and await further instructions or it may increment the content of a memory location by 1. In either case, step 2 takes at most a constant number of time units. In step 3, $n/2^t$ processors are used, one for each of the $n/2^t$ sets. Since it was assumed that a single processor can find the minimum of two elements in c time units, step 3 is executed with $n/2^t$ processors in c time units.

Since step 1 is performed only once, step 2, log n + 1 times, and step 3, log n times, the total time used is

$$1 + c'(\log n + 1) + c\log n = O(\log n).$$

The maximum number of processors in use during any unit of time is n/2, which occurs in the first iteration of step 3. Since these processors can be reassigned for later use, n/2 processors suffice to execute the entire algorithm. In the case where n is not necessarily a power of two, the procedure above can be modified to find the minimum of n elements in time O(log n) with $\lceil n/2 \rceil$ processors.

There is a way to solve this problem with asymptotically fewer processors. ALGORITHM MIN (2) below is a modification of

ALGORITHM MIN (1) and can be used to find the minimum of n elements in time $O(\log n)$ using only $\lceil n/\log n \rceil$ processors. Note first that a single processor can be used to find the minimum of k elements, $x(1), \ldots, x(k)$, in $c' \cdot (k-1)$ time units for some constant c'. Let $m(1) = \min \{x(1), x(2)\}$. For $t=2, \ldots, k-1$, let

$$m(t) = \min \{m(t-1), x(t+1)\}.$$

Then

$$m(k-1) = \min \{x(1), \ldots, x(k)\}.$$

Since for $t=1, \ldots, k-1$, $m(t)$ can be calculated from $m(t-1)$ and $x(t+1)$ in c' time units, for some constant c', $m(k-1)$ can be computed from $x(1), \ldots, x(k)$ in time $c' \cdot (k-1)$.

ALGORITHM MIN (2)

Assume that n elements are stored in a 1 x n array A.

Find $\min \{A(1), \ldots, A(n)\}$ in the following way.

1. Partition the n elements into $\lceil n/\log n \rceil$ sets $S_1, \ldots, S_{\lceil n/\log n \rceil}$ of at most $\lceil \log n \rceil$ elements each. Assign one processor to each set, S_i, to compute $m(i)$, a minimum element of S_i.
2. Use ALGORITHM MIN (1) to find m, the minimum of the $\lceil n/\log n \rceil$ elements, $m(1), \ldots, m(\lceil n/\log n \rceil)$. Then $m = \min \{A(1), \ldots, A(n)\}$.

In step 1, each of $\lceil n/\log n \rceil$ processors must find the minimum of at most $\lceil \log n \rceil$ numbers, which can be done in time at most

$c' \cdot (\lceil \log n \rceil - 1)$ as in the discussion preceding the algorithm. Step 2 is an execution of ALGORITHM MIN (1) on a set with $\lceil n/\log n \rceil$ elements which has been shown to use time $O(\log \lceil n/\log n \rceil)$ and $\frac{1}{2}\lceil n/\log n \rceil$ processors. Thus the total time required for ALGORITHM MIN (2) is bounded above by $c'(\lceil \log n \rceil - 1) + c_1 \log \lceil n/\log n \rceil = O(\log n)$. The maximum number of processors used is $\lceil n/\log n \rceil$, which occurs in step 1.

Note that these algorithms can be modified to find the maximum of n numbers in time $O(\log n)$ with $\lceil n/\log n \rceil$ processors.

Determining Whether a Binary Vector Contains a One

Let $\vec{v} = [v(1), \ldots, v(n)]$ be an n-dimensional vector where $v(i) \in \{0,1\}$ for $i=1, \ldots, n$. It will be necessary in some later algorithms to determine if such a vector contains a one. This can be done in time $O(\log n)$ with $\lceil n/\log n \rceil$ processors using ALGORITHM MIN (2) in the following way.

1. Let "$\leq$" be the usual ordering of the integers. Find $M = \max_{\leq} \{v(1), \ldots, v(n)\}$ using ALGORITHM MIN (2).
2. The vector $\vec{v}$ contains a 1 if and only if $M = 1$.

Determining Transitive Closure of a Boolean Matrix

Let $G = (V,E)$ be a directed graph. The transitive closure of G is the graph $G^* = (V,E^*)$ where for $x, y \in V$, $\overrightarrow{(x,y)} \in E^*$ if and only if there is a path from i to j in G of length at least 0. Let A be an $n \times n$ Boolean matrix. Then A is the adjacency matrix of the graph

$G_A = (V_A, E_A)$, where $V_A = \{1, \ldots, n\}$ and

$$E_A = \{\overrightarrow{(i,j)} \mid i,j \in V \text{ and } A(i,j) = 1\}.$$

The <u>transitive closure of A</u> is A^*, the adjacency matrix of the graph G_A^*.

Let B and C be nxn Boolean matrices.

Define the sum of B and C to be the nxn matrix B + C,

where

$$(B+C)(i,j) = \max \{B(i,j), C(i,j)\}.$$

Define the product of B and C as the nxn matrix BC, where

$$(BC)(i,j) = \max \{\min \{B(i,k), C(k,j)\} \mid k = 1, \ldots, n\}.$$

The following theorem gives a way of calculating A^* from A.

<u>Theorem 1</u>. Let G = (V,E) be a graph where V = {1, . . . , n}. Let A be the adjacency matrix of G and I, the nxn identity matrix. Then for any nonnegative integer k, $(A+I)^k (i,j) = 1$ if and only if there is a path in G from i to j of length at most k, where the addition and multiplication are as defined above.

<u>Proof</u>: If t is a nonnegative integer, then $A^t(i,j) = 1$ if and only if there is a path in G, of length t, from i to j. (See [9], p. 151.) Since $(A+I)^k = A^k + A^{k-1} + \ldots + A+I$, the theorem follows. QED

Since the length of the shortest path in G between any two vertices is at most n-1, this implies that $A^* = (A+I)^{n-1}$ and further, that $A^* = A^k$ for $k \geq n-1$. In particular, $A^* = A^{2^{\lceil \log(n-1) \rceil}}$ and thus A^* can be obtained by squaring the matrix, A, $\lceil \log(n-1) \rceil$ times. This fact is the basis of a parallel algorithm, due to Hirschberg [11], which computes the transitive closure of a Boolean matrix in time $O(\log^2 n)$ using n^3 processors. The number of processors can be reduced to $n^2 \lceil n/\log n \rceil$ if ALGORITHM MIN (2) is used. Recently, Chandra [6] has developed a parallel algorithm which finds the product of two nxn matrices in time $O(\log n)$ using $n^{\log 7}$ processors. It is based on Strassen's algorithm [27] for sequential matrix multiplication. If A^* is computed by squaring the matrix, A, $\lceil \log(n-1) \rceil$ times, where the squaring is done by using Chandra's algorithm, then A^* can be computed from A in time $O(\log^2 n)$ using only $(\log n)n^{\log 7}$ processors. If A is a <u>symmetric</u> nxn Boolean matrix, the transitive closure can be found in time $O(\log^2 n)$ using only $n \lceil n/\log n \rceil$ processors. This is a consequence of Hirschberg's algorithm [11] for finding the connected components of an undirected graph and will be discussed in detail in Section 3.3.

Sorting

Let R be a total order of a set S. An array X(1), . . . , X(n) of elements of S is <u>sorted</u> if X(i)RX(j) for i = 1, . . . , n-1. A sorting algorithm, given a 1xn array X of elements X(1), . . . , X(n) of S, computes a 1xn array Y, where Y(1), . . . , Y(n) consists of

the elements X(1), . . . , X(n) in sorted form. Preparata [18] has developed an algorithm to do this in time clogn with n⌈log n⌉ processors, for some constant c.

Although they will not be used here, it is interesting to note that classes of sorting algorithms have been found by Preparata [18] and Hirschberg [12] which, for any given $\alpha>o$, sort n numbers in time $(c/\alpha)\log n$ with $n^{1+\alpha}$ processors. Those of Preparata have the practical advantage that there is no time unit during which two processors are requested to read the content of the same memory location.

Chapter 3

SPANNING TREES

In Section 3.2 of this chapter, a parallel algorithm is presented which finds a minimum spanning tree of a connected, undirected, weighted graph with n vertices in time $O(\log^2 n)$, using $n\lceil n/\log n\rceil$ processors. It is shown in Section 3.3 that this algorithm can be modified to find a spanning tree, the connected components, and the transitive closure of a connected, undirected graph within the same time and processor bounds. An algorithm for finding a directed spanning tree of a connected, undirected graph is presented in Section 3.4.

The algorithm presented here for finding a minimum spanning tree uses techniques from [11] and [32], and it appears in [26].

3.1 Definitions

A <u>tree</u> was defined in Section 2.2. Note that a tree with n vertices has n-1 edges. Let $G = (V,E)$ be a connected, undirected graph. A <u>spanning tree</u> for G is a tree $T = (V,E')$ where $E' \subseteq E$. G is a <u>weighted</u> graph if there is a function $w : E \rightarrow \mathbb{R}$. The <u>weight matrix</u> of G is the nxn matrix W, where for $i,j \in V$,

$$W(i,j) = \begin{cases} w(i,j), & \text{if } (i,j) \in E \\ \infty, & \text{otherwise.} \end{cases}$$

(Here, ∞ is a symbol which has the property that $x < \infty$ for every real number x.)

A minimum spanning tree (MST) of the weighted graph G is a spanning tree $T = (V,E')$ of G such that $\sum_{e \in E'} w(e)$ is minimal among all spanning trees of G.

3.2 The Minimum Spanning Tree Algorithm

Let $G = (V,E)$ be a connected, undirected, weighted graph, with weight function w. Assume that the vertices of G are labeled by the integers 1 through n, that is, $V = \{1,2, ... , n\}$. Let W be the weight matrix of G. An algorithm is presented here which finds, from W, the adjacency matrix, M, of a MST of G. The construction used is described below. It will be shown that it is possible to carry out the construction, that the construction does result in a MST of G, and that the construction can be executed by the unbounded parallel model in time $O(\log^2 n)$ with $n\lceil n/\log n\rceil$ processors.

The notation used in the construction below will be used through the remainder of Chapter 3.

Construction of a Minimum Spanning Tree

A MST of G will be constructed inductively as follows.

1. For each $i \in V$, let $d_0(i) = i$. Let G_0 be the graph (V,ϕ).
2. If $d_0(i) \neq d_0(j)$ for some pair $i,j \in V$, it will be shown that it is possible to choose, for each $i \in V$, an edge $E_1(i)$ of G, incident with i, such that (a) $w(E_1(i))$ is minimal among all edges of G incident with i and (b) the undirected graph $G_1 = (V,T_1)$, where $T_1 = \{E_1(i) \mid i \in V\}$,

contains no cycles. Define a relation "$\equiv_1$" on V by $i \equiv_1 j$ if and only if there is a path in G_1 joining i and j. Then $\equiv_1$ is an equivalence relation on V and therefore partitions V into equivalence classes. Choose one vertex in each equivalence class as a representative. Let $d_1(i)$ be the representative of the equivalence class of $\equiv_1$ which contains i. For $i \in V$, let

$$[i]_1 = \{j \in V \mid d_1(i) = d_1(j)\}.$$

Then the set $S_1 = \{[i]_1 \mid d_1(i) = i\}$ is the collection of equivalence classes of V under the relation $\equiv_1$.

3. Assume inductively that for some $t \geq 1$, G_t, T_t, S_t and, for i = 1, . . . , n, $d_t(i)$ and $[i]_t$ have all been defined. If $d_t(i) \neq d_t(j)$ for some pair $i, j \in V$, it will be shown that it is possible to choose, for each $i \in V$ with $d_t(i) = i$, an edge $E_{t+1}(i)$ such that (a) $w(E_{t+1}(i))$ is minimal among all edges of G which join a vertex in $[i]_t$ to a vertex in another equivalence class in S_t and (b) the undirected graph $G_{t+1} = (V, T_{t+1})$, where

$$T_{t+1} = T_t \cup \{E_{t+1}(j) \mid j \in V \text{ and } d_t(j) = j\}$$

contains no cycles. Define a relation $\equiv_{t+1}$ on V by $i \equiv_{t+1} j$ if and only if there is a path in G_{t+1} joining i and j. Then $\equiv_{t+1}$ is an equivalence relation on V and therefore partitions V into equivalence classes. Choose one vertex in each equivalence class as a representative. For $i \in V$, let $d_{t+1}(i)$ be the representative of the equivalence class

of $\equiv_{t+1}$ which contains i. For $i \in V$ let

$$[i]_{t+1} = \{j \in V \mid d_{t+1}(i) = d_{t+1}(j)\}.$$

Then the set

$$S_{t+1} = \{[i]_{t+1} \mid d_{t+1}(i) = i\}$$

is the collection of equivalence classes of V under the relation $\equiv_{t+1}$.

It will be shown in Theorem 2 that for each $t \geq 0$ such that $d_t(u) \neq d_t(v)$ for some $u, v \in V$, there is for each $i \in V$, with $d_t(i) \neq i$, an edge $E_{t+1}(i)$ of G which satisfies the conditions stated in the construction. Let t^* be the smallest index for which $d_{t^*}(i) = d_{t^*}(j)$ for all $i, j \in V$. *Theorem 3 shows that the graph* G_{t^*} is a MST of G.

Assume that G has more than one vertex. For $i \in V$, let

$$m_1(i) = \min \{w(i,j) \mid j \in V\}.$$

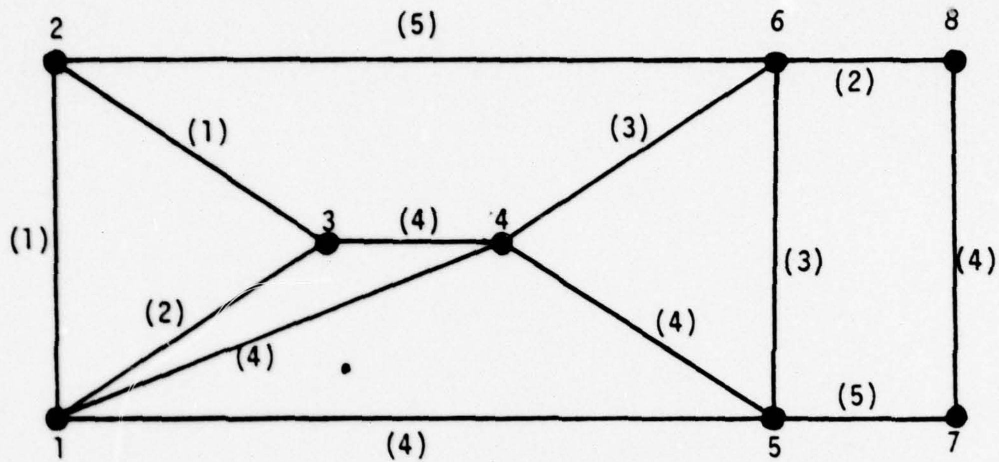
Since G is connected, $m_1(i) < \infty$. Define a function $C_1 : V \rightarrow V$ by

$$C_1(i) = \min \{j \mid j \in V \text{ and } w(i,j) = m_1(i)\}.$$

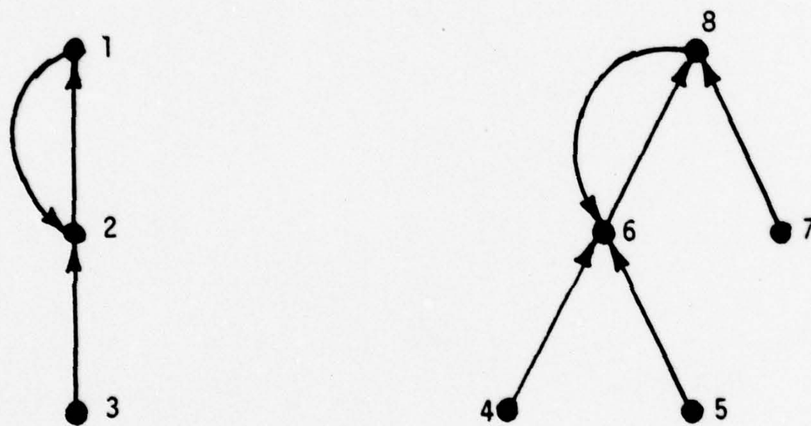
Let $G(C_1)$ be the directed graph (V, E'') where $E'' = \{\overrightarrow{(i, C_1(i))} \mid i \in V\}$. *Figure 1 shows* $G(C_1)$ *for a given weighted graph G.*

Lemma 1. Every cycle of $G(C_1)$ has length two.

Proof: Let $v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k$ be a cycle of $G(C_1)$ where $C_1(v_r) = v_{r+1}$ for $1 \leq r \leq k-1$ and $v_1 = v_k$. Then $2 \leq k \leq n$. By definition of C_1,



(a) The graph G



(b) The graph $G(C_1)$ for G

Figure 1. Construction of $G(C_1)$ from G

$$w(\overrightarrow{v_1,v_2}) \leq w(\overrightarrow{v_{k-1},v_k}) \leq w((\overrightarrow{v_{k-2},v_{k-1}}) \leq \ . \ . \ . \leq w(\overrightarrow{v_1,v_2}).$$

Thus all edges in the cycle have equal weight. Again by definition of C_1, we have the following inequalities. If k is even,

$$v_2 \leq v_k \leq v_{k-2} \leq \ . \ . \ . \leq v_2$$

which means k = 2. If k is odd,

$$v_2 \leq v_k \leq \ . \ . \ . \leq v_3 \leq v_1 \leq v_{k-1} \leq \ . \ . \ . \leq v_2$$

which is impossible if $k \geq 2$. QED

Theorem 2. Using the notation in the construction, for each $t \geq 0$, such that $d_t(u) \neq d_t(v)$ for some $u, v \in V$, and for each $i \in V$, with $d_t(i) = i$, there is an edge $E_{t+1}(i)$ of G satisfying conditions (a) and (b) in the construction.

Proof: For t = 0 and $i \in V$, let $E_1(i) = (i, C_1(i))$ where C_1 is the function defined preceding Lemma 1. Then by definition of C_1, $E_1(i)$ is an edge of G incident with i and $w(E_1(i))$ is minimal over all edges of G incident with i. Let $G_1 = (V, T_1)$ be the undirected graph where $T_1 = \{E_1(i) \mid i \in V\}$. If G_1 contains a cycle $v_1, \ldots, v_k$ then $v_1 = v_k$, $k \geq 4$, $C_1(v_1) = v_2$ or v_{k-1} and for r = 2, . . ., k-1, $C_1(v_r) = v_{r+1}$ or v_{r-1}. Without loss of generality, assume $C_1(v_1) = v_2$. If $C_1(v_r) = v_{r+1}$ for r = 1, . . . k-1 then $v_1 \rightarrow v_2 \rightarrow \ . \ . \ . \rightarrow v_k$ is a cycle of length at least 3 of the directed graph $G(C_1)$, contradicting Lemma 1. Otherwise, there is an index s, $2 \leq s \leq k$, such that $C_1(v_s) = v_{s-1}$. But then for $s \leq r \leq k$, $C_1(v_r)$ must be v_{r-1}, by definition of G_1.

This is a contradiction, since $C_1(v_k) = C_1(v_1) = v_2 \neq v_{k-1}$. Thus G_1 contains no cycles. (See Figure 2(a).)

Assume inductively that for some $t \geq 0$, such that $d_t(u) \neq d_t(v)$ for some $u, v \in V$, $E_t(i)$ has been chosen, for all $i \in V$ with $d_t(i) = i$, to satisfy (a) and (b) of the construction. Then the graph $G_t = (V,T_t)$ contains no cycles. Let $G' = (V',E')$ be an undirected, weighted graph whose vertices are the connected components of G_t. Each connected component of G_t corresponds to an equivalence class $[i]_t$ in S_t. Label the connected component of G_t corresponding to $[i]_t$ by $d_t(i)$. Then $V' = \{i \in V \mid d_t(i) = i\}$. For each $i, j \in V'$, let (i,j) be an edge in E' if and only if there is an edge of G joining a vertex of $[i]_t$ to a vertex in $[j]_t$. Define a weight function w' for $(i,j) \in E'$ by $w'(i,j) = \min \{w(u,v) \mid u \in [i]_t, v \in [j]_t\}$. G' is connected since G is. (See Figure 2(b).)

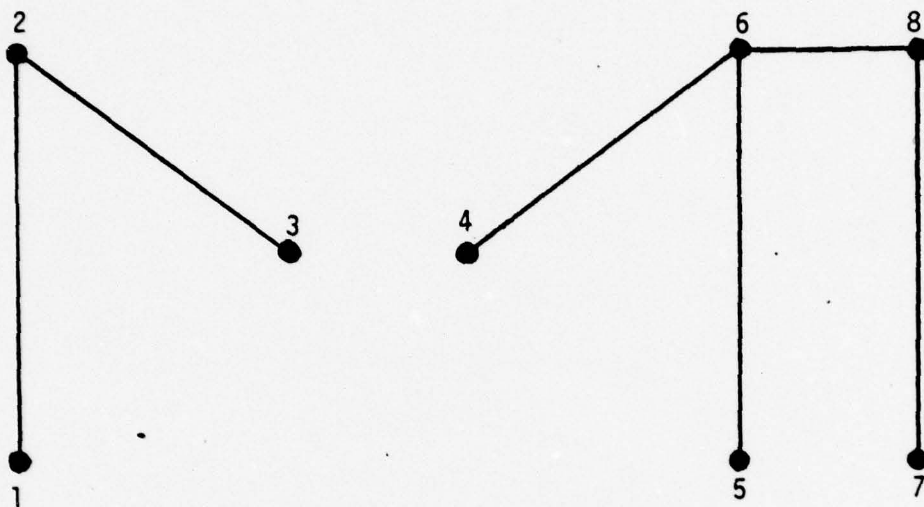
Define a function m_{t+1} on V' by

$$m_{t+1}(i) = \min \{w'(i,j) \mid j \in V'\}.$$

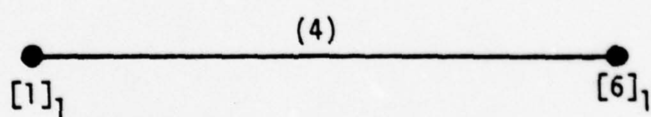
Since G_t is connected and contains more than one vertex, $m_{t+1}(i) \neq \infty$. Define $C_{t+1} : V' \rightarrow V'$ by

$$C_{t+1}(i) = \min \{j \mid w'(i,j) = m_{t+1}(i)\}.$$

Then the undirected graph $G'_{t+1} = (V',T'_{t+1})$, where



(a) The graph G_1



(b) The graph G' for the case $t = 1$

Figure 2. The graphs G_1 and G' for the graph G of Figure 1(a)

$$T'_{t+1} = \{(i, C_{t+1}(i)) \mid i \in V'\},$$

contains no cycles. This follows as in the proof that G_1 contains no cycles by replacing C_1 by C_{t+1} and G_1 by G'_{t+1}.

For each $i \in V'$, let $E_{t+1}(i)$ be an edge of G of weight $w'(i, C_{t+1}(i))$ which joins a vertex of $[i]_t$ to a vertex of $[C_{t+1}(i)]_t$, with the stipulation that if $C_{t+1}(C_{t+1}(i)) = i$, then $E_{t+1}(i) = E_{t+1}(C_{t+1}(i))$. Let $G_{t+1} = (V, T_{t+1})$, where

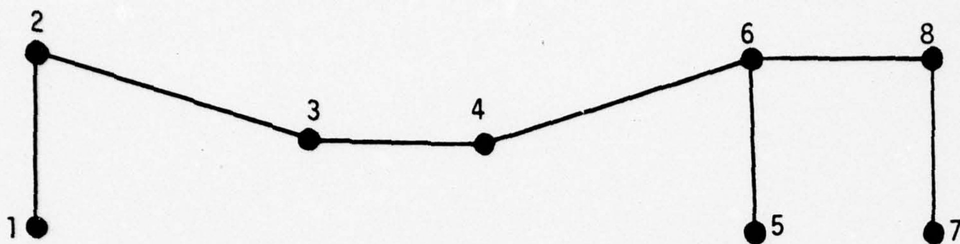
$$T_{t+1} = T_t \quad \{E_{t+1}(i) \mid d_t(i) = i\}.$$

(See Figure 3.) Since G_{t+1} is acyclic if and only if G_t and G'_{t+1} are acyclic, by the induction hypothesis, G_{t+1} is acyclic. Further, $E_{t+1}(i)$ is an edge of minimum weight over all edges which join a vertex of equivalence class $[i]_t$ to a vertex in another equivalence class in S_t. QED

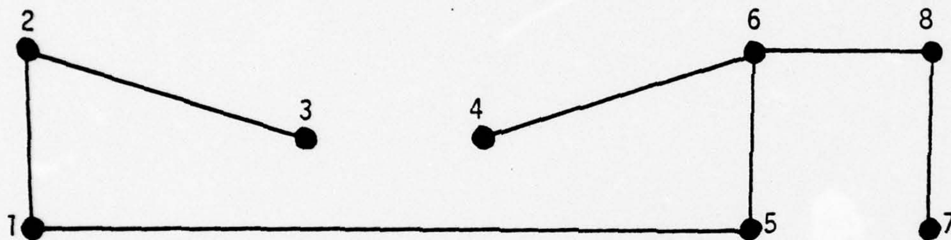
Theorem 3. Let t^* be the smallest index such that $d_{t^*}(i) = d_{t^*}(j)$ for all $i, j \in V$. Then G_{t^*} is a MST of G.

Proof: If $t^* = 0$, then G has only one vertex and the theorem is true. Thus, assume $t^* > 0$. If $d_{t^*}(i) = d_{t^*}(j)$ for all $i, j \in V$, then S_{t^*} has only one equivalence class, so G_{t^*} is connected. By Theorem 2, G_{t^*} contains no cycles. Thus G_{t^*} is a spanning tree of G. It remains to verify minimality.

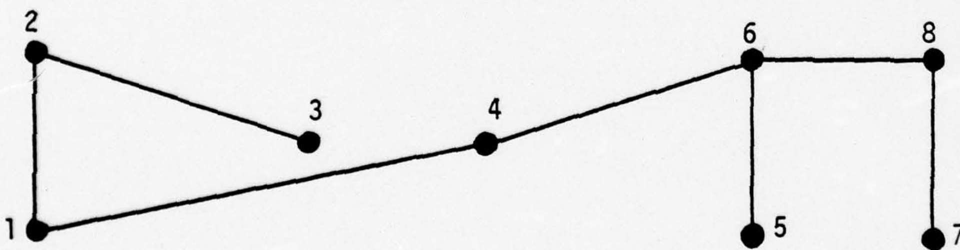
It will be shown by induction on t that for each t, with $1 \le t \le t^*$, the edges of G_t are contained in a MST of G. Let e be an edge of G of minimum weight. Then there is a MST of G which contains



(a) The graph G_2 if $E_{t+1}(d_t(1)) = E_{t+1}(d_t(6)) = (3,4)$



(b) The graph G_2 if $E_{t+1}(d_t(1)) = E_{t+1}(d_t(6)) = (1,5)$



(c) The graph G_2 if $E_{t+1}(d_t(1)) = E_{t+1}(d_t(6)) = (1,4)$

Figure 3. Three possible choices for G_2 from the Graph G of Figure 1(a)

e [21]. By definition, $G_1 = (V,T_1)$ must contain an edge of weight $w(e)$, thus at least one edge of G_1 is contained in a MST of G. Let T be a maximal subset of T_1 with the property that there is some MST of G containing the edges of T. Let $H = (V,E(H))$ be a MST of G with $T \subseteq E(H)$. If $T \neq T_1$, let e' be an edge of T_1 which is not in E(H). Since $e' \varepsilon T_1$, there is a vertex $v_1 \varepsilon V$, incident with e', such that e' is an edge of minimum weight among all edges of G incident with i. Adding e' to H creates a cycle $v_1, v_2, \ldots, v_k$ where $v_1 = v_k$ and $e' = (v_{k-1},v_k)$. Then $C_1(v_1) = v_{k-1}$. Let r be the smallest index for which $(v_r,v_{r+1}) \not\in T_1$. G_1 contains no cycles, $1 \leq r < k-1$. Then for $2 \leq s \leq r$, $C_1(v_s) = v_{s-1}$. Thus

$$w(e') \leq w(v_1,v_2) \leq \ldots \leq w(v_{r-1},v_r) \leq w(v_r,v_{r+1}).$$

then

$$H' = (V,E(H) \quad \{e'\} - \{(v_r,v_{r+1})\})$$

is a MST of G containing $T \quad \{e'\}$, contradicting the maximality of T.

Assume inductively that for some $t \geq 1$, the edges of G_t are contained in a MST of G. If G_t is connected, $t = t^*$ and G_t is a MST of G. Otherwise, let $G' = (V',E')$ be the undirected, weighted graph defined in the proof of Theorem 2, where the set

$$V' = \{i \varepsilon V \mid d_t(i) = i\}$$

represents the set of connected components of G_t. Let $C_{t+1} : V' \rightarrow V'$

be defined as in the proof of Theorem 2. Then the undirected graph $G'_{t+1} = (V', T'_{t+1})$, where

$$T'_{t+1} = \{(i, C_{t+1}(i)) \mid i \in V'\},$$

is a MST of G'. This follows as in the case where $t = 1$ by replacing G by G', C_1 by C_{t+1} and G_1 by G'_{t+1}. Recall that

$$T_{t+1} = T_t \cup \{E_{t+1}(i) \mid d_t(i) = i\}$$

where $E_{t+1}(i)$ is an edge of G joining a vertex of $[i]_t$ to a vertex of $[C_{t+1}(i)]_t$ and $w(E_{t+1}(i)) = w'(i, C_{t+1}(i))$. Then the edges in T_{t+1} are contained in a MST of G if and only if the edges in T_t are contained in a MST of G. (This can be proven by an argument similar to the one used to prove that the edges of G_1 are contained in a MST of G.) By the induction hypothesis, the edges in T_t are contained in a MST of G, thus, the edges of T_{t+1} are contained in a MST of G.

Thus, the edges of G_t^* are contained in a MST of G. Since G_t^* is acyclic and connected, G_t^* is a MST of G. QED

It follows from Lemma 2 below that $t^* \leq \lceil \log n \rceil$.

Lemma 2: For $t \geq 1$, S_t contains at most $n/2^t$ equivalence classes.

Proof: If $t = 1$, each equivalence class of S_1 has at least 2 vertices, so $|S_1| \leq n/2$ and the lemma is true. Assume for some $t \geq 1$ that $|S_t| \leq n/2^t$. Let $i \in V$ with $d_t(i) \neq i$. Edge $E_{t+1}(i)$ joins a vertex

u of $[i]_t$ with a vertex v in some distinct equivalence class $[j]_t$ of S_t. By definition of $\equiv_t$, every pair of vertices of $[i]_t$ can be joined by a path in G_t and similarly for every pair of vertices in $[j]_t$. Then every pair of vertices in $[i]_t \cup [j]_t$ can be joined by a path in G_{t+1}, since $E_{t+1}(j) \in T_{t+1}$. Thus $[i]_t \cup [j]_t$ is contained in a single equivalence class of S_{t+1}. Since $\equiv_t$ is a refinement of $\equiv_{t+1}$, this shows that every equivalence class in S_{t+1} is the union of at least two equivalence classes in S_t, thus $|S_{t+1}| \leq |S_t|/2 \leq n/2^{t+1}$. QED

It is shown below how to implement the construction of $G_t{}^*$ from G on the unbounded parallel model of computation. Initially, the weight matrix, W, of G is stored in the memory and at the end of the algorithm, the weight matrix, M, of $G_t{}^*$ is stored in the memory. The entire algorithm is presented and analyzed at the end of this section.

Construction of G_o and d_o

Since $G_o = (V,\phi)$ and $d_o(i) = i$ for all $i \in V$, the following steps compute d_o and the adjacency matrix of G_o.

1. For each pair $i, j \in V$, let $M(i,j) = 0$. Then M is the weight matrix of G_o.
2. For each $i \in V$ let $d_o(i) = i$.

Construction of G_1 and d_1

If $d_o(i) = d_o(j)$ for all $i, j \in V$ then G consists of a single vertex and G_o is a spanning tree of G. Otherwise, it was shown in Theorem 2 that G_1 can be calculated in the following way.

3. For each $i \varepsilon V$

(a) let $m_1(i) = \min \{W(i,j) \mid j \varepsilon V\}$

(b) let $C_1(i) = \min \{j \mid W(i,j) = m_1(i)\}$.

Then $G_1 = (V,T_1)$ where $T_1 = \{(i,C_1(i)) \mid i \varepsilon V\}$. In the construction of G_2, some operations are performed only for $i \varepsilon V$ such that $d_1(i) = i$. Thus d_1 must be computed. That is, for each equivalence class $[j]_1 \ \varepsilon S_1$ a vertex of $[j]_1$ must be chosen as a representative. Each equivalence class in S_1 corresponds to a connected component K of G_1. It is shown in Lemma 3 that each connected component K of G_1 contains exactly one pair of vertices i_K, j_K such that $C_1(i_K) = j_K$ and $C_1(j_K) = i_K$. Then $\min \{i_K, j_K\}$ is chosen to be the representative of the equivalence class of S_1 corresponding to K. In the example of Figure 1, $d_1(i) = 1$ for $i = 1,2,3$ and $d_1(j) = 6$ for $j = 4, \ldots, 8$.

<u>Lemma 3</u>. Let K be a connected component of G_1. Then K contains exactly one pair of vertices i,j such that $C_1(i) = j$ and $C_1(j) = i$.

<u>Proof</u>: For a vertex v of K, the vertices in the sequence, v, $C_1(v)$, $C_1(C_1(v))$, . . . , are all in K and cannot all be distinct. Then there must be a sequence

$$C_1^r(v),\ C_1^{r+1}(v),\ \ldots,\ C_1^s(v)$$

of distinct vertices of K for some r and s, where $C_1(C_1^s(v)) = C_1^r(v)$. If $s \neq r+1$, then the cycle

$$C_1^r(v) \rightarrow C_1^{r+1}(v) \rightarrow \ldots \rightarrow C_1^s(v) \rightarrow C_1^r(v)$$

is a cycle of length at least 3 in the directed graph $G(C_1)$ of Lemma 1, contradicting Lemma 1. Thus $C_1^r(v)$ and $C_1^s(v)$ are vertices of K such that $C_1(C_1^r(v)) = C_1^{r+1}(v) = C_1^s(v)$ and $C_1(C_1^s(v)) = C_1^r(v)$.

Assume i,j and i', j' are distinct pairs of vertices of K for which $C_1(i) = j$, $C_1(j) = i$, $C_1(i') = j'$ and $C_1(j') = i'$. Let $v_1, \ldots,$ v_k be a path in K where $v_1 \varepsilon \{i,j\}$ and $v_k \varepsilon \{i',j'\}$ and for $r = 2, \ldots,$ $k-1$, $v_r \notin \{i,j,i',j'\}$. Then it must be that $C_1(v_2) = v_1$ and $C_1(v_{k-1}) =$ v_k. By definition of G_1, for $s = 2, \ldots, k-1$, if $C_1(v_s) = v_{s-1}$, then $C_1(v_{s+1}) = v_s$. Thus, since $C_1(v_2) = v_1$, $C_1(v_{k-1}) = v_{k-2} \neq v_k$. This is a contradiction. Thus K has only one pair of vertices i,j with $C_1(i) = j$ and $C_1(j) = i$. QED

The function C_1 will be modified slightly to obtain a function C_1' where

$$C_1'(i) = \begin{cases} i, & \text{if } i \text{ is a representative of an equivalence class in } S_1 \\ C_1(i), & \text{otherwise.} \end{cases}$$

Since $C_1(C_1(i)) = i$ if and only if i or C(i) is an equivalence class representative, the edges $(i,C_1(i))$ and $(C_1(i), C_1(C_1(i)))$ of G_1 are identical and

$$T_1 = \{(i,C_1'(i)) \mid i \varepsilon V \text{ and } i \neq C_1'(i)\}.$$

The steps below compute the function C_1' and the weight matrix of G_1.

5. For each $i \in V$, let

$$C_1'(i) = \begin{cases} \min\{i, C_1(i)\}, \text{ if } C_1(C_1(i)) = i \\ C_1(i), \text{ otherwise.} \end{cases}$$

6. For each $i \in V$ with $C_1'(i) \neq i$, replace $M(i, C_1'(i))$ and $M(C_1'(i), i)$ by $W(i, C_1'(i))$. Then M is the weight matrix of G_1.

Lemma 4. For each $i \in V$, let $D_o(i) = C_1'(i)$ and for $k \geq 1$, let $D_k(i) = D_{k-1}(D_{k-1}(i))$. Then $d_1(i) = D_{\lceil \log(n-1) \rceil}(i)$ for all $i \in V$.

Proof: Let $i \in V$. Let $i = v_1, v_2, \ldots, v_r = d_1(i)$ be a simple path in G_1 joining i and $d_1(i)$. Then $r \leq n$. Since $C_1'(d_1(i)) = d_1(i) \neq v_{r-1}$, we must have $C_1'(v_{r-1}) = v_r$ and thus $C_1'(v_s) = v_{s+1}$ for $s = 1, \ldots, r-1$. Then for $s = 1, \ldots, r$,

$$D_o(v_s) = \begin{cases} v_{s+1}, \text{ if } s+1 < r \\ v_r, \text{ if } s+1 \geq r \end{cases}$$

and for $k \geq 1$,

$$D_k(v_s) = \begin{cases} v_{s+2^k}, \text{ if } s+2^k < r \\ v_r, \text{ if } s+2^k \geq r. \end{cases}$$

Thus

$$D_{\lceil \log(n-1)\rceil}(i) = D_{\lceil \log(n-1)\rceil}(v_1) = v_r = d_1(i),$$

since $1 + 2^{\lceil \log(n-1)\rceil} \geq n-1 \geq r-1$. QED

Then d_1 can be computed from C_1' in the following way.

7. Let $k = 0$ and for $i \in V$, let $D_0(i) = C_1'(i)$.

8. If $k \geq \log(n-1)$, then for $i \in V$ let $d_1(i) = D_k(i)$. Otherwise, increment k by 1.

9. For each $i \in V$, let $D_k(i) = D_{k-1}(D_{k-1}(i))$. Then return to step 8.

Construction of G_{t+1} and d_{t+1} from G_t and d_t

If $d_t(i) = d_t(j)$ for all $i,j \in V$, then G_t is a MST of G. Otherwise, it was shown in Theorem 2 that G_{t+1} can be constructed from G_t and d_t by finding, for each $i \in V$ such that $d_t(i) = i$, an edge $E_{t+1}(i)$ of G satisfying certain properties. Then $G_{t+1} = (V, T_{t+1})$ where $T_{t+1} = T_t \cup \{E_{t+1}(i) \mid d_t(i) = i\}$. For each $i \in V$ such that $d_t(i) = i$, define $m_{t+1}(i)$, as in the proof of Theorem 2, by

$$m_{t+1}(i) = \min\{w(u,v) \mid u \in [i]_t \text{ and for some } j \in V,\ v \in [j]_t \text{ and } i \neq j\}.$$

Since G is connected and G_t has more than one component, $m_{t+1}(i) < \infty$. For each $i \in V$ such that $d_t(i) = i$, define $C_{t+1}(i)$, as in the proof of Theorem 2, by

$$C_{t+1}(i) = \min \{j \in V \mid i \neq j,\ d_t(j) = j \text{ and for some } u \in [i]_t,\ v \in [j]_t,\ w(u,v) = m_{t+1}(i)\}.$$

Then by Theorem 2, for each $i \in V$ with $d_t(i) = i$, $E_{t+1}(i)$ is an edge of G of weight $m_{t+1}(i)$ which joins a vertex in $[i]_t$ to a vertex in $[C_{t+1}(i)]_t$. However, there may be several edges of G with this property and any one may be chosen as $E_{t+1}(i)$. (See Figure 3.) To have a precise algorithm, there must be a rule for choosing one edge from among all possible choices for $E_{t+1}(i)$. Assume $(i_1,j_1), \ldots, (i_k,j_k)$ are the edges of G which satisfy, for $r = 1, \ldots, k$, $i_r \in [i]_t$, $j_r \in [C_{t+1}(i)]_t$, and $w(i_r,j_r) = m_{t+1}(i)$. Let

$$h_{t+1}(i) = \min \{i_r \mid r = 1, \ldots, k\}$$

and

$$g_{t+1}(i) = \min \{j_r \mid r = 1, \ldots, k \text{ and } i_r = h_{t+1}(i)\}.$$

Then let $E_{t+1}(i) = (h_{t+1}(i), g_{t+1}(i))$. In the example of Figure 3, $E_1(1) = E_1(6) = (1,4)$.

The computation of $C_{t+1}(i)$ and $E_{t+1}(i)$, for each $i \in V$ with $d_t(i) = i$, is made precise below. The functions m'_{t+1}, a_{t+1}, and g'_{t+1}, which will be defined, are just intermediate steps in the calculation of m_{t+1}, h_{t+1}, g_{t+1}, and C_{t+1}.

12. For each $i \in V$,

(a) let $m'_{t+1}(i) = \min \{W(i,j) \mid j \in V \text{ and } d_t(j) \neq d_t(i)\}$

(b) let $a_{t+1}(i) = \min \{d_t(j) \mid j \in V,\ d_t(j) \neq d_t(i) \text{ and } W(i,j) = m'_{t+1}(i)\}$

(c) let $g'_{t+1}(i) = \min \{j \in V \mid d_t(j) = a_{t+1}(i) \text{ and } W(i,j) = m'_{t+1}(i)\}$.

13. For each $i \in V$ such that $d_t(i) = i$

(a) let $m_{t+1}(i) = \min \{m'_{t+1}(j) \mid d_t(j) = i,\ j \in V\}$

(b) let $C_{t+1}(i) = \min \{a_{t+1}(j) \mid j \in V,\ d_t(j) = i \text{ and } m'_{t+1}(j) = m_{t+1}(i)\}$

(c) let $h_{t+1}(i) = \min \{j \in V \mid d_t(j) = i \text{ and } m'_{t+1}(j) = m_{t+1}(i) \text{ and } a_{t+1}(j) = C_{t+1}(i)\}$

(d) let $g_{t+1}(i) = g'_{t+1}(h_{t+1}(i))$.

At this point, G_{t+1} has been found and its weight matrix can be constructed. It remains to calculate d_{t+1}. Let $G'_{t+1} = (V', T'_{t+1})$ where $V' = \{i \in V \mid d_t(i) = i\}$ and

$$T'_{t+1} = \{(i, C_{t+1}(i)) \mid i \in V \text{ and } d_t(i) = i\}.$$

Choose a vertex of V' from each connected component of G'_{t+1} as a representative. For each $i \in V'$, let $d'_{t+1}(i)$ be the representative of the connected component of V' containing i. The component representatives can be chosen and d'_{t+1} can be computed just as equivalence class representatives were chosen and d_1 computed in steps 5 through 9.

14. For each $i \in V$ with $d_1(i) = i$, let

$$C'_{t+1}(i) = \begin{cases} \min \{i, C_{t+1}(i)\}, & \text{if } C_{t+1}(C_{t+1}(i)) = i \\ C_{t+1}(i), & \text{otherwise.} \end{cases}$$

15. For each $i \in V$ with $d_t(i) = i$ and $C'_{t+1}(i) \neq i$, replace $M(h_{t+1}(i), g_{t+1}(i))$ and $M(g_{t+1}(i), h_{t+1}(i))$ by $W(h_{t+1}(i), g_{t+1}(i))$. Then M is the weight matrix of G_{t+1}.

16. Let $k = 0$ and for $i \in V$ with $d_t(i) = i$, let $D_0^{(t+1)}(i) = C'_{t+1}(i)$.

17. If $k \geq \log(n-1)$, then for each $i \in V$ with $d_t(i) = i$, let $d'_{t+1}(i) = D_k^{(t+1)}(i)$ and go on to step 19. Otherwise, increment k by 1.

18. For each $i \in V$ with $d_t(i) = i$, let

$$D_k^{(t+1)}(i) = D_{k-1}^{(t+1)}(D_{k-1}^{(t+1)}(i)).$$

Then return to step 16.

Let S be an equivalence class in S_{t+1}. Then S is the union of some equivalence classes in S_t. In step 14, an equivalence class $[k]_t \in S_t$, where $d_t(k) = k$, was chosen as a representative of the collection of all equivalence classes $[j]_t \in S_t$ such that $[j]_t \subseteq S$. In steps 15 through 17, d'_{t+1} was calculated so that for each $j \in V$ with $d_t(j) = j$, such that $[j]_t \subset S$, $d'_{t+1}(j) = k$. We now choose the vertex k to be the representative of the equivalence class S and compute $d_{t+1} : V \rightarrow V$ so that $d_{t+1}(i) = k$ for every $i \in S$.

19. For each $i \in V$, let $d_{t+1}(i) = d'_{t+1}(d_t(i))$. Note that if $d_t(i) = i$, then $d_{t+1}(i) = d'_{t+1}(i)$.

To complete the algorithm, there must be a way to decide for each $t \geq 0$ whether or not $d_t(i) = d_t(j)$ for all $i,j \in V$. This can be done in the following way. For $i = 1, \ldots, n-1$, let

$$z_t(i) = \begin{cases} 0, & \text{if } d_t(i) = d_t(i+1) \\ 1, & \text{otherwise.} \end{cases}$$

Then $d_t(i) \neq d_t(j)$ for some $i,j \in V$ if and only if the n-1 dimensional vector $[z_t(1), \ldots, z_t(n-1)]$ contains a 1. It was shown in Section 2.2 that this can be decided in time $O(\log(n-1))$ with n-1 processors.

The entire algorithm is presented below. After each step, the time and number of processors used to perform the step are given. Following the algorithm, there is an analysis of the time and processor complexities of the algorithm as a whole.

ALGORITHM MST

Let $G = (V,E)$ be a connected, undirected, weighted graph, where $V = \{1, \ldots, n\}$. Assume that the weight matrix, W, of G is stored in the memory. Compute the weight matrix, M, of a MST of G in the following way.

1. For each pair $i,j \in V$, let $M(i,j) = 0$. (This can be done in time $O(\log n)$ with $\lceil n^2/\log n \rceil$ processors.)
2. For each $i \in V$, let $d_0(i) = i$. (constant time, n processors)
3. If $d_0(i) = d_0(j)$ for all $i,j \in V$, stop, having found M. Otherwise, go on to step 4. (By the remarks preceding the algorithm, this can be done in time $O(\log n)$ with n-1 processors.)
4. For each $i \in V$,

(a) let $m_1(i) = \min \{W(i,j) \mid j \in V\}$

(b) let $C_1(i) = \min \{j \in V \mid W(i,j) = m_1(i)\}$. (For each $i \in V$, the minimum of at most n numbers must be found. Thus, by the results of Section 2.2, step 4 can be done in time $O(\log n)$ with $n\lceil n/\log n\rceil$ processors.)

5. For each $i \in V$, let

$$C_1'(i) = \begin{cases} \min \{i, C_1(i)\}, \text{ if } C_1(C_1(i)) = i \\ C_1(i), \text{ otherwise.} \end{cases}$$

(constant time, n processors)

6. For each $i \in V$ with $C_1'(i) \neq i$, replace $M(i,C_1'(i))$ and $M(C_1'(i),i)$ by $W(i, C_1'(i))$. (constant time, n processors)

7. Let $k = 0$. For $i \in V$, let $D_0(i) = C_1'(i)$. (constant time, n processors)

8. If $k \geq \log(n-1)$, then for $i \in V$, let $d_1(i) = D_k(i)$ and go on to step 10. Otherwise, increment k by 1. (constant time, n processors)

9. For each $i \in V$, let $D_k(i) = D_{k-1}(D_{k-1}(i))$. Then return to step 8. (constant time, n processors)

10. Let $t = 1$. (constant time, 1 processor)

11. If $d_t(i) = d_t(j)$ for all $i,j \in V$, stop, having found M. Otherwise, increment t by 1. (This can be done in time $O(\log n)$ with n-1 processors as in step 3.)

12. For each $i \in V$,

(a) let $m_{t+1}'(i) = \min \{W(i,j) \mid j \in V \text{ and } d_t(j) \neq d_t(i)\}$

(b) let $a_{t+1}(i) = \min \{d_t(j) \mid j \in V, d_t(j) \neq d_t(i)$ and $W(i,j) = m'_{t+1}(i)\}$

(c) let $g'_{t+1}(i) = \min \{j \in V \mid d_t(j) = a_{t+1}(i)$ and $W(i,j) = m'_{t+1}(i)\}$. (This can be done in time $O(\log n)$ with $n\lceil n/\log n\rceil$ processors, as in step 4.)

13. For each $i \in V$ such that $d_t(i) = i$,

(a) let $m_{t+1}(i) = \min \{m'_{t+1}(j) \mid j \in V, d_t(j) = i\}$

(b) let $C_{t+1}(i) = \min \{a_{t+1}(j) \mid j \in V, d_t(j) = i$ and $m'_{t+1}(j) = m_{t+1}(i)\}$

(c) let $h_{t+1}(i) = \min \{j \in V \mid d_t(j) = i, m'_{t+1}(j) = m_{t+1}(i)$ and $a_{t+1}(j) = C_{t+1}(i)\}$

(d) let $g_{t+1}(i) = g'_{t+1}(h_{t+1}(i))$. (This can be done in time $O(\log n)$ with $n\lceil n/\log n\rceil$ processors, as in step 12.)

14. For each $i \in V$ with $d_t(i) = i$, let

$$C'_{t+1}(i) = \begin{cases} \min \{i, C_{t+1}(i)\}, & \text{if } C_{t+1}(C_{t+1}(i)) = i \\ C_{t+1}(i), & \text{otherwise.} \end{cases}$$

(constant time, n processors)

15. For each $i \in V$ with $d_t(i) = i$ and $C'_{t+1}(i) \neq i$, replace $M(h_{t+1}(i), g_{t+1}(i))$ and $M(g_{t+1}(i), h_{t+1}(i))$ by $W(g_{t+1}(i), h_{t+1}(i))$.

(constant time, n processors)

16. Let $k = 0$. For each $i \in V$ with $d_t(i) = i$, let $D_0^{(t+1)}(i) = C'_{t+1}(i)$.

(constant time, n processors)

17. If $k \geq \log(n-1)$, then for each $i \in V$ with $d_t(i) = i$, let $d'_{t+1}(i) = D_k^{(t+1)}(i)$ and go on to step 19. Otherwise, increment k by 1. (constant time, n processors)

18. For each $i \in V$ with $d_t(i) = i$, let

$$D_k^{(t+1)}(i) = D_{k-1}^{(t+1)}(D_{k-1}^{(t+1)}(i)).$$

Then return to step 17. (constant time, n processors)

19. For each $i \in V$, let $d_{t+1}(i) = d'_{t+1}(d_t(i))$. Then return to step 11. (constant time, n processors)

Steps 8 and 9 of ALGORITHM MST are iterated at most $\lceil \log(n-1) \rceil + 1$ times. Thus, steps 1 through 10 can be performed in time $O(\log n)$ with $n\lceil n/\log n \rceil$ processors. Steps 11 through 19 of ALGORITHM MST are iterated at most $\lceil \log n \rceil$ times, by Lemma 2. In each iteration, steps 17 and 18 are performed at most $\lceil \log(n-1) \rceil + 1$ times, so one iteration of steps 11 through 19 takes time at most $O(\log n)$ and $n\lceil n/\log n \rceil$ processors. Thus the entire algorithm can be performed in time

$$O(\log n) + \lceil \log n \rceil \, O(\log n) = O(\log^2 n)$$

with $n\lceil n/\log n \rceil$ processors.

The best serial algorithms for finding a minimum spanning tree of G have time complexity $O(n^2)$ [20], using the adjacency matrix as input, and $O(m \log\log m)$ [32], where $m = |E|$, using an adjacency list representation of G. Both of these algorithms have time complexity $O(n^2)$ if $m = O(n^2)$. Thus ALGORITHM MST has a speedup of $O(n^2/\log^2 n)$ over the best serial

algorithm and an efficiency of O(1/logn). This indicates that there is relatively little waste incurred by using $n\lceil n/\log n\rceil$ processors to achieve the speedup of $O(n^2/\log^2 n)$.

3.3 Applications of the MST Algorithm

Spanning Trees

Let G = (V,E) be a connected, undirected graph, where V = {1, . . ., n}. Let A be the adjacency matrix of G. Then a spanning tree of G can be computed from A in the following way.

1. For each pair $i,j \in V$, let

$$W(i,j) = \begin{cases} \infty, & \text{if } A(i,j) = 0 \\ 1, & \text{otherwise} \end{cases}$$

2. Execute ALGORITHM MST using W as input. The nxn matrix M, computed in this way, is the adjacency matrix of a spanning tree of G.

Since step 1 can be done in time O(logn) with $\lceil n^2/\log n\rceil \le n\lceil n/\log n\rceil$ processors, a spanning tree of G can be found in time $O(\log^2 n)$ with $n\lceil n/\log n\rceil$ processors.

In the algorithm presented in Chapter 4 for finding the biconnected components of a connected, undirected graph G, the first step is to find a spanning tree, S, for G. For this problem, the adjacency matrix of S is not the most efficient representation of S. Let V = {1, . . ., n}. Then S can be represented by an (n-1) x 2 array T where $T(i,j) \in V$ and the set

$$\{(T(i,1),T(i,2)) \mid i = 1, . . ., n-1\}$$

is the set of edges of S. By using this representation of S, the number of processors used to find the biconnected components of G can be decreased by a factor of n.

ALGORITHM MST can be modified to compute an n x 2 array H where for some $k \in V$, $H(k,1) = H(k,2) = 0$, and for $i \in V$, $i \neq k$, and $j = 1,2$, $H(i,j) \in V$ and

$$\{(H(i,1),H(i,2)) \mid i \in V, i \neq k\}$$

is the set of edges of a spanning tree of G. To do this, at the beginning of ALGORITHM MST, let $H(i,j) = 0$ for $i \in V$ and $j = 1,2$. Replace step 15 by $15'$ below.

$15'$. For each $i \in V$ with $d_t(i) = i$ and $C'_{t+1}(i) \neq i$, replace $H(i,1)$ by $h_{t+1}(i)$ and $H(i,2)$ by $g_{t+1}(i)$.

If for some i, $d_t(i) = i$ and $C'_{t+1}(i) \neq i$, then $d_{t+1}(i) \neq i$. Thus for each i, there is at most one edge to be stored as $(H(i,1),H(i,2))$.

Assume now that the n x 2 array H has been computed for a spanning tree S of G. Compute the (n-1) x 2 array T in the following way.

1. For each $i \in V$, let

$$y(i) = \begin{cases} 0, & \text{if } H(i,1) = H(i,2) = 0 \\ i, & \text{otherwise.} \end{cases}$$

2. Sort the numbers $y(1), \ldots, y(n)$ in decreasing order to obtain an array $z(1), \ldots, z(n)$.

3. For $i = 1, \ldots, n-1$, let $T(i,1) = H(z(i),1)$ and $T(i,2) = H(z(i),2)$. Then

$$\{(T(i,1),T(i,2)) \mid i = 1, \ldots, n-1\}$$

is the set of edges of a spanning tree of G.

Steps 1 and 3 can be done in constant time with n processors. Step 2 can be done in time $O(\log n)$ with $n\lceil \log n \rceil$ processors. Thus calculation of the $(n-1) \times 2$ array T can be incorporated into ALGORITHM MST without increasing the time and processor complexity. This is summarized in the theorem below.

Theorem 4. Let $G = (V,E)$ be a connected, undirected graph, where $V = \{1, \ldots, n\}$. There is an algorithm which computes, from the adjacency matrix of G, an $(n-1) \times 2$ array T, where $T(i,j) \in V$ for $i = 1, \ldots, n-1$, $j = 1,2$, and

$$\{(T(i,1),T(i,2)) \mid i = 1, \ldots, n-1\}$$

is the set of edges of a spanning tree for G. The algorithm has time complexity $O(\log^2 n)$ and processor complexity $n\lceil n/\log n \rceil$.

Connected Components and Transitive Closure

Let $G = (V,E)$ be an undirected graph, where $V = \{1, \ldots, n\}$. A spanning forest for G is a forest $S = (V,E')$ where $E' \subseteq E$ and every vertex of G is incident with some edge in E'. ALGORITHM MST

can be modified to find the adjacency matrix of a spanning forest of G in time $O(\log^2 n)$ with $n\lceil n/\log n\rceil$ processors [26]. Under this modification, if t^* is the smallest index for which G_{t^*} is a spanning forest of G, then for $i,j \in V$, $d_{t^*}(i) = d_{t^*}(j)$ if and only if i and j are in the same connected component of G. The connected components of G are computed in this way by Hirschberg in [11].

Theorem 5. Let $G = (V,E)$ be an undirected graph, where $V = \{1, \ldots, n\}$. There is an algorithm which computes, from the adjacency matrix of G, a $1 \times n$ array D, where, for $i,j \in V$, $D(i) = D(j)$ if and only if i and j are in the same connected component of G. The algorithm has time complexity $O(\log^2 n)$ and processor complexity $n\lceil n/\log n\rceil$.

Proof: This follows by combining ALGORITHM MIN(2) with the algorithm presented in [11]. QED

Corollary 1. Let $G = (V,E)$ be an undirected graph, where $V = \{1, \ldots, n\}$. Let $f_G = 1$, if G is connected, 0, otherwise. There is an algorithm which computes f_G, from the adjacency matrix of G, in time $O(\log^2 n)$ with $n\lceil n/\log n\rceil$ processors.

Proof: Compute the $1 \times n$ array D of Theorem 5. Then G is connected if and only if $D(i) = D(j)$ for all $i,j \in V$. It was shown in the discussion preceding ALGORITHM MST that it is possible to decide whether or not all entries of a given n dimensional vector are the same in time $O(\log n)$ with $n-1$ processors. Thus, if

$$f_G = \begin{cases} 1, & \text{if } D(i) = D(j) \text{ for all } i,j \in V \\ 0, & \text{otherwise,} \end{cases}$$

then f_G can be computed in time $O(\log^2 n)$ with $n\lceil n/\log n\rceil$ processors from the adjacency matrix of G. QED

Corollary 2. Let $G = (V,E)$ be an undirected graph, where $V = \{1, . . ., n\}$ and let A be the adjacency matrix of G. There is an algorithm which computes, from A, the transitive closure of A in time $O(\log^2 n)$ with $n\lceil n/\log n\rceil$ processors.

Proof: First compute, from A, the 1xn array D of Theorem 5. Then for each $i,j \in V$, let

$$A^*(i,j) = \begin{cases} 1, & \text{if } D(i) = D(j) \\ 0, & \text{otherwise.} \end{cases}$$

Then A^* is the transitive closure of A. QED

3.4 Directed Spanning Trees

For a given directed graph $D = (V,E)$ let $U(D)$ be the corresponding undirected graph, $U(D) = (V,E')$, where

$$E' = \{(i,j) \mid \overrightarrow{(i,j)} \in E \text{ or } \overrightarrow{(j,i)} \in E\}.$$

Let $G = (V,E)$ be an undirected graph, where $V = \{1, . . ., n\}$. A directed spanning forest of G is a directed forest T, where $U(T)$ is a spanning forest of G. If G is connected, a directed spanning tree of G is a directed, rooted tree, (T,r), where $U(t)$ is a spanning tree of G.

Assume G is connected. The algorithms in Chapters 6 and 7, for finding the bridges of G and a cycle of G, begin by finding a directed spanning tree of G. This can be done by using a modification of ALGORITHM MST and an algorithm for finding the transitive closure of a directed forest. It will be shown in Lemma 13 of Chapter 6 that the transitive closure of a directed forest can be found in time $O(\log n)$ with n^2 processors.

<u>Theorem 6</u>. Let $t_{TC}(n)$ and $p_{TC}(n)$ be the time and processor complexities, respectively, of an algorithm which finds the transitive closure of a directed forest with n vertices. Let A be the adjacency matrix of a connected, undirected graph $G = (V,E)$ with n vertices. There is an algorithm which computes, from A, a function $F : V \to V$ such that the graph

$$H = (V, \{\overrightarrow{(i,F(i))} \mid i \in V, i \neq F(i)\})$$

is a directed spanning tree of G, in time $O(\log^2 n + \log n \cdot t_{TC}(n))$ with $\max\{n\lceil n/\log n\rceil, p_{TC}(n)\}$ processors.

<u>Proof</u>: We use the notation of Section 3.2 and let $F_1(i) = C_1'(i)$ for all $i \in V$. The directed graph

$$G_1' = (V, \{\overrightarrow{(i,F_1(i))} \mid i \in V, i \neq F_1(i)\})$$

is a directed spanning forest of G_1. Assume inductively that for some $t \geq 1$, a function $F_t:V\to V$ has been defined so that the graph

$$G_t' = (V, \{\overrightarrow{(i,F_t(i))} \mid i \in V, i \neq F_t(i)\})$$

is a directed spanning forest of G_t. Define $F_{t+1}:V\to V$, so that the graph

$$G'_{t+1} = (V, \{\overrightarrow{(i,F_{t+1}(i))} \mid i \in V,\ i \neq F_{t+1}\})$$

is a directed spanning forest of G_{t+1}, in the following way. (See Figure 4.)

(a) For each $i \in V$ with $d_t(i) = i$ and $C'_{t+1}(i) \neq i$, let $F_{t+1}(h_{t+1}(i)) = g_{t+1}(i)$.

(b) For each $j \neq d_t(j)$ on the path in G'_t from $h_{t+1}(d_t(j))$ to $d_t(j)$, $F_{t+1}(F_t(j)) = j$.

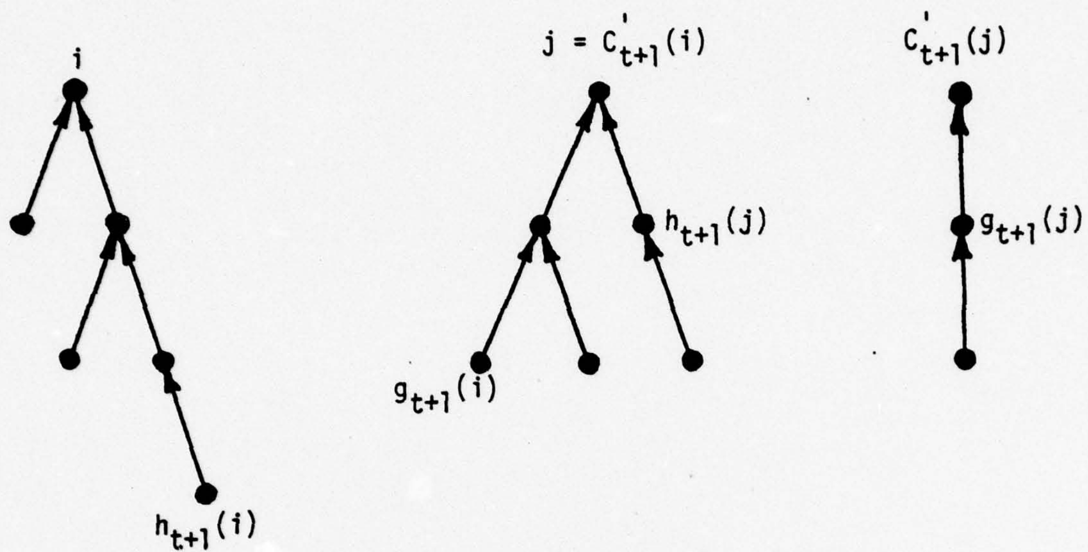
(c) For all other vertices, $F_{t+1}(i) = F_t(i)$.

To implement the construction of G'_{t+1} from G_t, there must be a way to decide, for a given $j \in V$, whether j satisfies the conditions in (b). Let M_t be the adjacency matrix of G'_t and let M^*_t be the transitive closure of M_t. Then $j \in V$ is on the path in G'_t from $h_{t+1}(d_t(j))$ to $d_t(j)$ if and only if $M^*_t(h_{t+1}(d_t(j)),j) = 1$ and $M^*_t(j,d_t(j)) = 1$. Thus, the following modifications of ALGORITHM MST will compute F_1 and will compute, for $t \geq 1$, F_{t+1} from F_t. Replace step 6 by step $6'$.

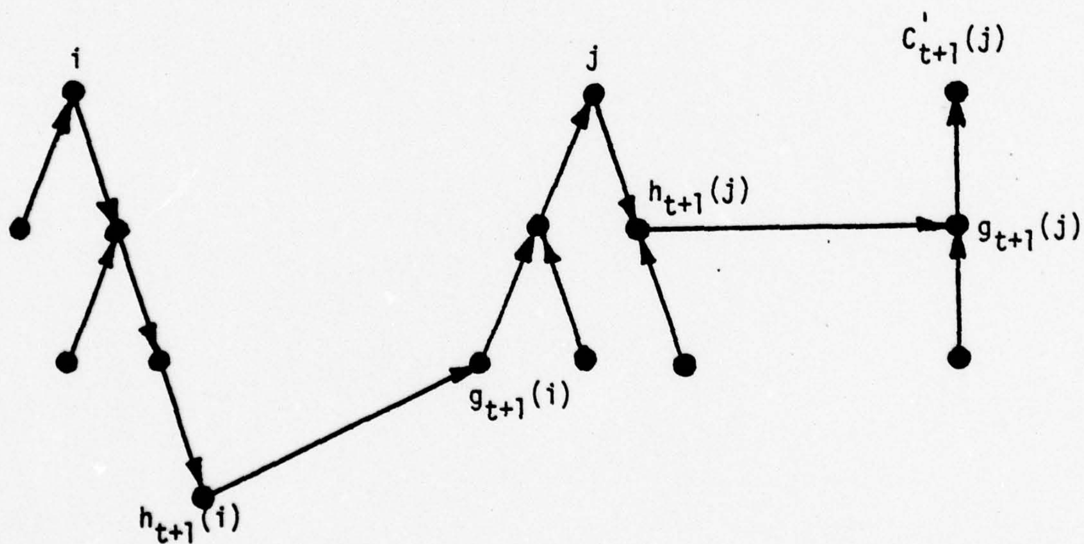
> $6'$. For each $i \in V$, let $F(i) = C'_1(i)$. Then $F = F_1$. For each $i \in V$ with $i \neq F(i)$, replace $M(i,F(i))$ by 1. Then M is the adjacency matrix of G'_1.

Insert step $6\frac{1}{2}$ between steps $6'$ and 7.

> $6\frac{1}{2}$. Find M^*. Then M^* is the transitive closure of the adjacency matrix of G'_1.



(a) The graph G_t'



(b) The graph G_{t+1}'

Figure 4. An example of the construction of G_{t+1}' from G_t'

Replace step 15 by step 15'.

15.' (i) For each $i \in V$ with $d_t(i) = i$ and $C'_{t+1}(i) \neq i$, replace $F(h_{t+1}(i))$ by $g_{t+1}(i)$. For each $i \in V$ with $i \neq d_t(i)$, such that $M^*(h_{t+1}(d_t(i)),i) = 1$ and $M^*(i,d_t(i)) = 1$, replace $F(j)$ by i, where $j = F(i)$. Then $F_{t+1} = F$.

(ii) For each $i \in V$, with $i \neq F(i)$, replace $M(i,F(i))$ by 1. Then M is the adjacency matrix of G'_{t+1}.

Insert step $15\frac{1}{2}$ between steps 15' and 16.

$15\frac{1}{2}$. Find M^*. Then M^* is the transitive closure of the adjacency matrix of G'_{t+1}.

When the modified algorithm halts, the function F is such that the graph

$$G'_{t^*} = (V, \{\overrightarrow{(i,F(i))} \mid i \in V, i \neq F(i)\})$$

is a directed spanning tree of G. The root of G'_{t^*} is the vertex $r \in V$ for which $F(r) = r$. The nxn matrix M stored in the memory is the weight matrix of G'_{t^*}.

Step 15' can be done in constant time with n processors and step $15\frac{1}{2}$ *can be done in time* $t_{TC}(n)$ *with* $p_{TC}(n)$ processors. Thus in the modified algorithm, one iteration of steps 11 through 19 requires time $O(\log n) + t_{TC}(n)$ and $\max\{n\lceil n/\log n\rceil, p_{TC}(n)\}$ processors. Since steps 11 through 19 are iterated at most $\lceil \log n\rceil$ times, and since the rest of the algorithm can be done in time $O(\log^2 n)$ with $\lceil n^2/\log n\rceil$ processors, the entire modified algorithm can be performed in time $O(\log^2 n + \log n \cdot t_{TC}(n))$ with $\max\{n\lceil n/\log n\rceil, p_{TC}(n)\}$ processors. QED

Chapter 4

BICONNECTED COMPONENTS

Let $G = (V,E)$ be an undirected graph. A vertex $a \varepsilon V$ is an <u>articulation point</u> of G if there are vertices $u, v \varepsilon V$, distinct from a, such that every path in G joining u and v contains a. A graph is <u>biconnected</u> if it is connected and contains no articulation points. A <u>biconnected component</u> of G is a subgraph H of G with the property that H is biconnected and if K is any biconnected subgraph of G such that $H \subseteq K$ then $H = K$. If $V' \subseteq V$, let

$$E(V') = \{(x,y) \varepsilon E \mid x \varepsilon V' \text{ and } y \varepsilon V'\}.$$

Note that if $B = (V',E')$ is a biconnected component of G, then $e' = E(V')$. Otherwise, there would be an edge $e \varepsilon E(V') - E'$ and then $K = (V',E' \cup \{e\})$ would be a biconnected subgraph of G such that $B \subseteq K$, but $B \neq K$.

In this chapter, an algorithm will be presented which finds the biconnected components of a connected, undirected graph G in time $O(\log^2 n)$ with $n^2\lceil n/\log n\rceil$ processors. The algorithm will find the sets of vertices of the biconnected components of G, since the set of vertices of a biconnected component determine it completely. The following lemmas concerning biconnected components will be needed.

<u>Lemma 5</u>. If $B_1 = (V_1,E_1)$ and $B_2 = (V_2,E_2)$ are two distinct biconnected components of G then V_1 V_2 contains at most one vertex.

Proof: See [1], p. 181. QED

Lemma 6. Vertices i and j of a graph G are in a common biconnected component of G if and only if $i = j$, (i,j) is an edge of G, or i and j lie on a common cycle of G.

Proof: See [9], p. 27. QED

Let $G = (V,E)$ be a connected, undirected graph. The biconnected components of G will be found in the following way. First, a spanning tree, S, of G is found. It will be shown, in Lemma 8, that each edge of S determines a biconnected component of G and, in Lemma 9, that every biconnected component of G contains an edge of S. Next, a relation R is defined on V, and computed by the algorithm, so that iRj if and only if i and j are in a common biconnected component of G. It is shown in Lemma 8 that if (i,j) is an edge of S, and

$$V' = \{k \in V \mid kRi \text{ and } kRj\}$$

then $B = (V', E(V'))$ is the biconnected component of G containing (i,j). In this way, the biconnected component of G determined by (i,j) is computed.

Let R be the relation on V which is defined above. Note that R is reflexive and symmetric, but not necessarily transitive. For each $i \in V$, construct a graph $G_i = (V,E_i)$, where E_i consists of all edges in E except those incident with vertex i. Let $G_i^* = (V,E_i^*)$ be the transitive closure of G_i. Then the following theorem holds.

Theorem 7. Let $G = (V,E)$ be a connected, undirected graph. For $i,j \in V$, iRj if and only if $(i,j) \in E_k^*$ for every $k \in V$, distinct from i and j.

(Note in particular that iRi for all $i \in V$ and iRj for all $(i,j) \in E$. The proof of this theorem relies on the following lemma and its corollary.)

Lemma 7. Let $G = (V,E)$ be a connected, undirected graph. For distinct vertices x, y, and z in V, if xRy and yRz and xRz then x, y and z are contained in a common biconnected component of G.

Proof: If $(x,y) \in E$, let C_1 be the graph $C_1 = (\{x,y\}, \{(x,y)\})$. Otherwise, let C_1 be a cycle of G containing x and y. Similarly, define C_2 and C_3 as graphs consisting of either a single edge, (y,z) or (x,z), respectively, or a cycle of G containing y and z, or x and z, respectively. For $i = 1,2,3$, the graphs C_i exist, by Lemma 6. Note that for $i = 1,2$, or 3, if u and v are vertices of C_i, and k is a vertex of V, distinct from u and v, there is a path in C_i joining u and v which does not contain k.

Claim: The graph $B = C_1 \cup C_2 \cup C_3$ is biconnected. To see this, let u and v be distinct vertices of B, such that u is a vertex of C_i and v is a vertex of C_j for some $i,j \in \{1,2,3\}$. Since $C_i \cap C_j \neq \phi$, let $w \in C_i \cap C_j$. Let k be a vertex of B, distinct from u and v. If $w = u$ or $w = v$, then u and v are both in C_i or C_j and there is a path joining u and v in C_i or C_j (and thus in B) which does not contain k. If w is distinct from u and v, let p_i be the path of C_i, joining u and w, which does not contain k, and p_j, the path of C_j, joining w and v, which does not contain k. Then $p_i \cup p_j$ is

a path in B joining u and v which does not contain k. Thus, B is biconnected and is therefore contained in some biconnected component, B', of G. Since x, y, and z are vertices of B, they all lie in the biconnected component B'. QED

Corollary. Let G = (V,E) be a connected, undirected graph. For vertices u, v, x, and y of V, such that $u \neq v$, assume uRv and xRu, yRu, xRv, and yRv. Then u, v, x, and y are contained in a common biconnected component of G.

Proof: By Lemma 7, u, v, and x are contained in a biconnected component, $B_1 = (V_1, E(V_1))$, of G and u, v, and y are contained in a biconnected component, $B_2 = (V_2, E(V_2))$, of G. Since $V_1 \cap V_2 = \{u,v\}$ and $u \neq v$, by Lemma 5, $V_1 = V_2$ and so $B_1 = B_2$. Thus, u, v, x and y are vertices of B_1. QED

Proof of Theorem 7: If i = j, the theorem is true. Assume $i \neq j$. If iRj, let $B = (V_B, E(V_B))$ be the biconnected component of G containing i and j. Let k be a vertex of V, distinct from i and j. Since B is biconnected, B is connected and there is a path in B joining i and j. If $k \notin V_B$ this path cannot contain k. If $k \in V_B$, k is not an articulation point of B and thus there is a path in B joining i and j which does not contain k. Thus $(i,j) \in E_k^*$.

Conversely, assume $(i,j) \in E_k^*$ for all $k \in V$, distinct from i and j. Let $B = (V_B, E_B)$ be a biconnected component of G containing i.

Let

$$B' = B \cup \{p \mid p \text{ is a simple path in } G \text{ joining } i \text{ and } j\}.$$

Let x and y be distinct vertices of B'. Let k be any vertex of B', distinct from x and y. If x, y ε V_B, there is a path in B joining x and y which does not contain k. If neither x nor y is in V_B, assume x lies on the simple path p_x and y, on the simple path p_y, joining i and j. There is a path in p_x, joining either x and i or x and j, which does not contain k. Similarly, there is a path in p_y, joining either y and i or y and j, which does not contain k. Since (i,j) ε E_k^*, there is a simple path in B', joining i and j, which does not contain k. These paths can be combined to produce a path in B' joining x and y which does not contain k. A similar argument shows that if x ε V_B, but y $\notin$ V_B, there is a path in B' joining x and y which does not contain k. Thus B' is a biconnected subgraph of G containing B. Then $B = B'$ and, in particular, j ε B and iRj. QED

<u>Lemma 8</u>. Let S be a spanning tree of the connected, undirected graph G = (V,E) and let e = (x,y) be an edge of S. Then e is in some biconnected component of G. Further, if

$$V' = \{z \in V \mid zRx \text{ and } zRy\}$$

then $B' = (V', E(V'))$ is the biconnected component of G containing e.

<u>Proof</u>: By Theorem 7, since e = (x,y) ε E, xRy, that is, x and y

are contained in a common biconnected component, $B = (W,E(W))$, of G. Since $x,y \in W$, $e = (x,y) \in E(W)$.

For the second part of the theorem, note that since xRx, yRy, and xRy, x and y are in V' and thus $(x,y) \in E(V')$. To see that $B' = (V',E(V'))$ is biconnected, let $u, v \in V'$. By the corollary to Lemma 7, uRv, thus, by Theorem 7, $(u,v) \in E^*_k$ for all vertices k of V', distinct from u and v. Thus B' has no articulation points. To see that B' is a biconnected component, let $B'' = (V'',E(V''))$ be a biconnected component of G containing B'. Let w be an element of V''. Then since $x,y \in V' \subseteq V''$, wRx and wRy. Thus $w \in V'$, so $V' = V''$ and $B' = B''$. QED

Lemma 9. Let S be a spanning tree of the connected, undirected graph $G = (V,E)$. Then every biconnected component of G contains an edge of S.

Proof: Let $B' = (V',E(V'))$ be a biconnected component of G. If V has only one vertex, S contains no edges and the lemma is true. Otherwise, since G is connected, V' has more than one vertex and $E(V')$ is nonempty. If $E(V')$ contains no edge of S, let $e = (u,v)$ be an edge of $E(V')$. Then adding edge e to S creates a cycle, C, in S which contains e. Let (x,y) be an edge of C, distinct from (u,v). By Lemma 6, xRu, yRu, xRv, and yRv. Thus, since uRv, it follows from the corollary to Lemma 7 that u, v, x, and y are contained in a common biconnected component, $B'' = (V'',E(V''))$ of G. Since $\{u,v\} \subseteq V' \cap V''$, $B'' = B'$, by Lemma 5. Thus, x and y are in V' and $(x,y) \in E(V')$, which contradicts the assumption that $E(V')$ contains no edge of S. QED

Lemmas 8 and 9 verify that the biconnected components of G can be computed by the method described at the beginning of this chapter. For each biconnected component, $B = (V_B, E(V_B))$, of G, an n-dimensional vector $\vec{B}$ will be computed so that for each $j \in V$, $\vec{B}(j) = 1$ if and only if $j \in V_B$. Before the algorithm is presented, one other problem must be handled. If S is a spanning tree of G and e and e' are edges of S, e and e' may be in the same biconnected component of G. If, for every edge in S, the biconnected component associated with that edge is computed, some biconnected component of G will be computed more than once. This duplication can be eliminated in the following way.

Let T be an (n-1)x2 array, where the set

$$\{(T(i,1),T(i,2)) \mid i = 1, \ldots, n-1\}$$

is the set of edges of a spanning tree, S, of G. For $i,j \in \{1, \ldots, n-1\}$, the edges $(T(i,1),T(i,2))$ and $(T(j,1),T(j,2))$ are in the same biconnected component of G if and only if $T(i,1)\ R\ T(j,1)$, $T(i,2)\ R\ T(j,2)$, $T(i,1)\ R\ T(j,2)$, and $T(i,2)\ R\ T(j,1)$. The necessity of this condition is a consequence of the definition of R and the sufficiency follows from the corollary to Lemma 7. For $i = 1, \ldots, n-1$, let $m(i)$ be the smallest index j for which the edge $(T(i,1), T(i,2))$ is in the same biconnected component of G as the edge $(T(j,1),T(j,2))$. Then the set of edges,

$$Q = \{(T(i,1),T(i,2)) \mid i = 1, \ldots, n-1 \text{ and } m(i) = i\},$$

has the property that each biconnected component of G contains exactly

one edge of Q. From the function m, the number, K, of connected components of G can be computed and a Kxn array, B, can be constructed where $B(i,j) = \vec{B_i}(j)$ and the n-dimensional vectors, $\vec{B_1}, \ldots, \vec{B_K}$, represent the biconnected components of G.

For i = 1, . . ., n-1, let

$$y(i) = \begin{cases} i, & \text{if } m(i) = i \\ 0, & \text{otherwise.} \end{cases}$$

Let z(1), . . ., z(n-1) be the elements y(1), . . ., y(n-1) in decreasing order. The largest index k for which $z(k) \neq 0$ is K, the number of biconnected components of G and

$$Q = \{(T(z(i),1),T(z(i),2)) \mid i = 1, \ldots, K\}.$$

For i = 1, . . ., K and $j \in V$, let

$$\vec{B_i}(j) = \begin{cases} 1, & \text{if } jRT(z(i),1) \text{ and } jRT(z(k),2) \\ 0, & \text{otherwise.} \end{cases}$$

Then the vectors, $\vec{B_1}, \ldots, \vec{B_K}$ represent the biconnected components of G. The entire algorithm is presented below, followed by an analysis of the time and processor complexity.

ALGORITHM BICONNECTED COMPONENTS

Let A be the adjacency matrix of a connected, undirected graph $G = (V,E)$, where $V = \{1, . . ., n\}$. Find the biconnected components of G in the following way.

1. Compute an $(n-1) \times 2$ array T so that the set

$$\{(T(i,1),T(i,2)) \mid i = 1, . . ., n-1\}$$

is the set of edges of a spanning tree of G.

2. For each $k \in V$, construct an nxn matrix M_k, where for $i,j \in V$,

$$M_k(i,j) = \begin{cases} 0, \text{ if } i = k \text{ or } j = k \\ A(i,j), \text{ otherwise.} \end{cases}$$

(Then M_k is the adjacency matrix of the graph G_k, obtained from G by removing every edge incident with i.)

3. For each $k \in V$, compute M_k^*, the transitive closure of M_k.
4. For each pair $i,j \in V$, let

$$R(i,j) = \min \{M_k^*(i,j) \mid k \in V, k \neq i,j\}.$$

(Then $R(i,j) = 1$ if and only if iRj.)

5. For $i = 1, . . ., n-1$, let

$$\begin{aligned} m(i) = \min \{j \mid j=1, . . ., n-1 \text{ and } & R(T(i,1),T(j,1)) \\ &= R(T(i,2),T(j,2)) = R(T(i,1),T(j,2)) \\ &= R(T(i,2),T(j,1)) = 1\}. \end{aligned}$$

(Then for $i = 1, \ldots, n-1$, $m(i) = m(j)$ if and only if the edges $(T(i,1),T(i,2))$ and $(T(j,1),T(j,2))$ are in a common biconnected component of G.)

6. For $i = 1, \ldots, n-1$, let

$$y(i) = \begin{cases} 0, & \text{if } m(i) \neq i \\ i, & \text{otherwise.} \end{cases}$$

7. Sort the array $y(1), \ldots, y(n-1)$ in decreasing order to obtain a sorted array $z(1), \ldots, z(n-1)$.

8. Let $K = \max \{i \mid i=1, \ldots, n-1 \text{ and } z(i) \neq 0\}$. (Then K is the number of biconnected components of G and each biconnected component of G contains an edge $(T(j,1),T(j,2))$ where $j \in \{1, \ldots, K\}$.)

9. For $i = 1, \ldots, K$ and $j \in V$, let

$$\overrightarrow{B_i}(j) = \begin{cases} 1, & \text{if } R(j,T(z(i),1)) = R(j,T(z(i),2)) = 1 \\ 0, & \text{otherwise.} \end{cases}$$

(Then the 1xn arrays $\overrightarrow{B_1}, \ldots, \overrightarrow{B_K}$ represent the biconnected components $B_1', \ldots, B_K'$ of G, where $B_i' = (V_1', E(V_1'))$ and $V_1' = \{j \in V \mid \overrightarrow{B_i}(j) = 1\}$.)

Notice that G is biconnected if and only if $K = 1$. Thus, this algorithm also determines whether a given connected, undirected graph is biconnected.

Analysis of Time and Processor Requirements

1. By Theorem 4 of Section 3.3, step 1 can be done in time $O(\log^2 n)$ with $n\lceil n/\log n\rceil$ processors.

2. Partition the set $\{(i,j,k) \mid i,j,k \in V\}$ into $\lceil n^3/\log n\rceil$ sets of at most $\lceil \log n\rceil$ numbers each. Assign one processor to each set X to compute $M_k(i,j)$ for each of the triples $(i,j,k) \in X$. Then step 2 can be done in time $O(\log n)$ with $\lceil n^3/\log n\rceil$ processors.

3. By Corollary 2 of Theorem 5, the transitive closure of the adjacency matrix of an undirected graph can be found in time $O(\log^2 n)$ with $n\lceil n/\log n\rceil$ processors. Thus step 3 can be done in time $O(\log^2 n)$ with $n^2\lceil n/\log n\rceil$ processors.

4. In step 4, the minimum of at most n numbers must be found for each pair $(i,j) \in V \times V$. This can be done in time $O(\log n)$ with $n^2\lceil n/\log n\rceil$ processors.

5. In step 5, for each $i \in V$, the minimum of at most n numbers must be found. This can be done in time $O(\log n)$ with $n\lceil n/\log n\rceil$ processors.

6. Step 6 can be done in constant time with n processors.

7. As discussed in Section 2.2, n-1 numbers can be sorted in time $O(\log(n-1))$ with $(n-1)\lceil \log(n-1)\rceil$ processors.

8. In step 8, the maximum of at most n numbers can be found in time $O(\log n)$ with $n\lceil n/\log n\rceil$ processors.

9. Step 9 can be done in constant time with $K \cdot n \leq n^2$ processors.

Thus, the total time used by ALGORITHM BICONNECTED COMPONENTS is $O(\log^2 n)$ and the maximum number of processors used is $n^2\lceil n/\log n\rceil$, in step 3.

The fastest serial algorithm for finding the biconnected components of a graph G, with n vertices and m edges, uses an adjacency list representation of G and has time complexity $O(m)$[1]. If $m = O(n^2)$, this algorithm has time complexity $O(n^2)$. If the adjacency matrix of G is used as input, it can be shown (see Theorem 13) that, for some constant c, at least cn^2 entries of the adjacency matrix of G must be examined to determine whether or not G is biconnected. Thus any serial algorithm which determines biconnectivity from the adjacency matrix must have time complexity at least cn^2. Since ALGORITHM BICONNECTED COMPONENTS has time complexity $O(\log^2 n)$ and processor complexity $O(n^3/\log n)$, a speedup of $O(n^2/\log^2 n)$ is achieved over the best serial algorithm, with an efficiency of $O(1/n\log n)$. Although the speedup is large, the efficiency indicates that a relatively large amount of waste is incurred by using $O(n^3/\log n)$ processors to achieve the speedup.

Chapter 5

DOMINATORS

A rooted graph, (G,r), is a directed graph $G = (V,E)$ with a distinguished vertex r such that there is a path in G from r to i for every $i \in V$. For vertices i and j of V, i is a dominator of j in (G,r) if every path from r to j in G contains i. In particular, for every i in V, r and i are dominators of i. For vertices i,j and k of V, whenever i is a dominator of j and j is a dominator of k, then i is a dominator of k and, if $k = i$, then $i = j$. Thus, the relation R on V, defined for i and j in V by iRj if and only if i is a dominator of j, is a partial order of V. For each vertex j of V, let $S_j = \{i \in V \mid iRj\}$. Then R, restricted to the set S_j, is a total order of S_j. For each vertex j of V, distinct from r, the vertex $\max_R \{i \in S_j \mid i \neq j\}$ is called the immediate dominator of j in (G,r). Since R is antisymmetric, the immediate dominator of j is unique. The dominator relation R on V can be represented by a directed, rooted tree, (T_D,r) with vertex set V in which the father of each vertex j, distinct from r, is the immediate dominator of j in (G,r). Then i is a dominator of j in (G,r) if and only if i is an ancestor of j in T_D. T_D is called the dominator tree of G.

In this chapter, a parallel algorithm is presented which finds, for a directed, rooted graph (G,r), with $|V| = n$, an nxn array D, where, for $i,j \in V$,

$$D(i,j) = \begin{cases} 1, & \text{if } i \text{ is a dominator of } j \text{ in } (G,r) \\ 0, & \text{otherwise.} \end{cases}$$

D will be called the <u>dominator matrix</u> of (G,r). In addition, the algorithm computes a function $F : V \to V$, where the directed graph,

$$T_D = (V, \{\overrightarrow{(i,F(i))} \mid i \in V, i \neq F(i)\}),$$

is the dominator tree of (G,r). The time complexity of the algorithm is $O(\log^2 n)$ and the processor complexity is $(\log n)\, n^{1+\log 7}$.

Let (G,r) be a rooted graph where $G = (V,E)$ and $V = \{1, . . ., n\}$. The dominator relationship on V can be computed in the following way. For each vertex k of V, distinct from r, let G_k be the directed graph obtained from G by removing all edges which leave the vertex k. For each vertex k of V, let G_k^* be the transitive closure of the graph G_k.

<u>Lemma 10</u>. For $i, j \in V$, iRj if and only if $i = r$, $i = j$, or $(r,j) \notin G_i^*$.

<u>Proof</u>: If iRj, $i \neq j$, and $i \neq r$, then every path in G from r to j contains i. Thus, if all edges leaving i are removed from G, there is no path in G from r to j, that is, $(r,j) \notin G_i^*$. Conversely, if $i = r$ or $i = j$ then iRj. If $(r,j) \notin G_i^*$, there is no path from r to j in G_i. Thus, any path in G from r to j must include i, so iRj. QED

Thus, the following steps compute the dominator matrix, D, of (G,r) from r and the adjacency matrix, A, of G.

1. For each $k \varepsilon V$ with $k \neq r$, and each pair $i,j \varepsilon V$, let

$$M_k(i,j) = \begin{cases} 0 \text{ if } i = k \\ A(i,j), \text{ otherwise.} \end{cases}$$

Then M_k is the adjacency matrix of G_k.

2. For each $k \varepsilon V$ with $k \neq r$, find M_k^*, the transitive closure of M_k.

3. For each $i,j \varepsilon V$, let

$$D(i,j) = \begin{cases} 1, \text{ if } i = r \text{ or } i = j \\ 1, \text{ if } i \neq r \text{ and } M_i^*(r,j) = 0 \\ 0, \text{ otherwise.} \end{cases}$$

To compute the dominator tree of (G,r) let $F(j)$ be the immediate dominator of j in (G,r) for each vertex j of V, distinct from r, and let $F(r) = r$. If the nxn array, D, has been calculated then, given $u, v \varepsilon S_j$, a processor can calculate $\max_R \{u,v\}$ in constant time, since

$$\max_R \{u,v\} = \begin{cases} v \text{ if } R(u,v) = 1 \\ u, \text{ otherwise.} \end{cases}$$

Thus, ALGORITHM MIN (2) of Section 2.2 can be used to find $F(j)$. The entire algorithm is summarized and analyzed below.

ALGORITHM DOMINATORS

Let (G,r) be a rooted graph, where $G = (V,E)$ and $V = \{1, . . ., n\}$. Given r and the adjacency matrix, A, of G, find the dominator matrix, D, of (G,r) and for each $r \in V$, with $i \neq r$, find $F(i)$, the immediate dominator of i in (G,r) in the following way.

1. For each $k \in V$, with $k \neq r$, and each pair $i,j \in V$, let

$$M_k(i,j) = \begin{cases} 0, \text{ if } i = k \\ A(i,j), \text{ otherwise.} \end{cases}$$

(constant time, n^3 processors)

2. For each $k \in V$, with $k \neq r$, find M_k^*. (It was mentioned in Section 2.2 that the transitive closure of a directed graph can be found in time $O(\log^2 n)$ with $(\log n)n^{\log 7}$ processors.)

3. For each pair $i,j \in V$, let

$$D(i,j) = \begin{cases} 1, \text{ if } i = r \text{ or } i = j \\ 1, \text{ if } i \neq r \text{ and } M_i^*(r,j) = 0 \\ 0, \text{ otherwise.} \end{cases}$$

(constant time, n^2 processors)

4. For each $j \in V$ with $j \neq r$, let

$$F(j) = \max_R \{i \in V \mid i \neq j \text{ and } D(i,j) = 1\}.$$

Let $F(r) = r$. (This can be done using ALGORITHM MIN (2) of Section 2.2 in time $O(\log n)$ with $n\lceil n/\log n\rceil$ processors.)

ALGORITHM DOMINATORS has time complexity $O(\log^2 n)$ and processor complexity $(\log n)n^{1+\log 7}$. The best serial algorithm for this problem uses the adjacency list representation of G and has time complexity $O(n\log n + m)$, where m is the number of edges of G [29]. If $m = O(n^2)$, this algorithm has time complexity $O(n^2)$. It can be shown (see Theorem 14) that any serial algorithm which computes the dominators of G, from the adjacency matrix of G, has time complexity at least cn^2, for some constant c. Comparing the parallel and serial algorithms, the parallel algorithm achieves a speedup of $O(n^2/\log^2 n)$ with an efficiency of $O(1/n^{\log 7-1}\log^3 n)$. Although the speedup is large, there is much waste in using so many processors to achieve it.

Chapter 6

BRIDGES

Let $G = (V,E)$ be a connected, undirected graph with n vertices. An edge e of E is a bridge of G if the graph $G' = (V,E-\{e\})$ is not connected. A graph is bridge connected if it is connected and contains no bridges. A bridge connected component of G is a maximal bridge connected subgraph of G.

In Section 6.1, a straightforward algorithm is presented which finds the bridges of G in time $O(\log^2 n)$ with $n^2\lceil n/\log n\rceil$ processors. In Section 6.3, a more involved algorithm is presented to find the bridges of G in time $O(\log^2 n)$ using only $n^2\lceil \log n\rceil$ processors. It is shown in Section 6.4 that the bridge connected components of G can also be found in time $O(\log^2 n)$ with $n^2\lceil \log n\rceil$ processors. Both algorithms for finding the bridges of a graph are based on the following lemma. (See, for example, [30].)

Lemma 11. Let $G = (V,E)$ be a connected, undirected graph. If $e \in E$ is a bridge of G, then e is contained in every spanning tree of G.

Proof: Let S be the set of edges of a spanning tree for G. If $e \notin S$ then removal of e from G does not disconnect G. Thus e is not a bridge of G. QED

6.1 First Bridge Algorithm

Let $G = (V,E)$ be a connected, undirected graph, where $V = \{1, . . ., n\}$. Let $S = (V,E')$ be a spanning tree of G, where

$E' = \{e, \ldots, e_{n-1}\}$. By Lemma 11, if e is a bridge of G, then $e \in E'$. Thus, the bridges of G can be found by testing each edge of E' to determine whether or not it is a bridge of G. For $i = 1, \ldots, n-1$, let G_i be the graph obtained from G by removing edge e_i. Then e_i is a bridge of G if and only if G_i is not connected. Let the function $f : E' \to \{0,1\}$ be defined for $e_i \in E'$ by

$$f(e_i) = \begin{cases} 1, & \text{if } e_i \text{ is a bridge of G} \\ 0, & \text{otherwise.} \end{cases}$$

Then f can be computed from the adjacency matrix, A, of G in the following way.

1. Compute an $(n-1) \times 2$ array T so that the set

$$E' = \{(T(i,1),T(i,2)) \mid i = 1, \ldots, n-1\}$$

is the set of edges of a spanning tree, S, of G.

2. For $k = 1, \ldots, n-1$ and for each pair $i,j \in V$, let

$$M_k(i,j) = \begin{cases} 0, & \text{if } (i,j) = (T(k,1),T(k,2)) \\ A(i,j), & \text{otherwise.} \end{cases}$$

Then M_k is the adjacency matrix of the graph G_k.

3. For $i = 1, \ldots, n-1$, let

$$f(i) = \begin{cases} 0, \text{ if } G_k \text{ is connected} \\ 1, \text{ otherwise.} \end{cases}$$

Then $f(i) = 1$ if and only if $(T(i,1),T(i,2))$ is a bridge of G.

Let K be the number of bridges of G. Once f has been computed, the bridges of G can be stored in a K x 2 array, B, so that the set

$$\{(B(i,1),B(i,2)) \mid i = 1, \ldots, K\}$$

is the set of bridges of G. The following steps compute K and the K x 2 array, B.

4. For $i = 1, \ldots, n-1$, let

$$y(i) = \begin{cases} i, \text{ if } f(i) = 1 \\ 0, \text{ otherwise.} \end{cases}$$

5. Sort the numbers $y(1), \ldots, y(n-1)$ in decreasing order to obtain the array $z(1), \ldots, z(n-1)$.
6. Let $m = \min \{i \mid i = 1, \ldots, n-1, z(i) = 0\}$. Let $K = m-1$. Then K is the number of bridges of G.
7. For $i = 1, \ldots, K$ and $j = 1,2$, let $B(i,j) = T(z(i),j)$. Then

$$\{(B(i,1),B(i,2)) \mid i = 1, \ldots, K\}$$

is the set of bridges of G.

Notice that G is bridge connected if and only if K = 0. Thus this procedure for determining the bridges of G also decides whether or not G is bridge connected. The entire algorithm is summarized and analyzed below.

ALGORITHM BRIDGES (1)

Let $G = (V,E)$ be a connected, undirected graph, where $V = \{1, \ldots, n\}$. Using the adjacency matrix, A, of G as input, find K, the number of bridges of G, and a $K \times 2$ array, B, so that the set $\{(B(i,1),B(i,2)) \mid i = 1, \ldots, K\}$ is the set of bridges of G, in the following way.

1. Compute an $(n-1) \times 2$ array T where the set

$$\{(T(i,1),T(i,2)) \mid i = 1, \ldots, n-1\}$$

is the set of edges of a spanning tree of G. (By Theorem 4 of Section 3.3, this can be done in time $O(\log^2 n)$ with $n\lceil n/\log n\rceil$ processors.)

2. For $k = 1, \ldots, n-1$, and for each pair $(i,j) \in V$, let

$$M_k(i,j) = \begin{cases} 0, & \text{if } i = T(k,1) \text{ and } j = T(k,2) \\ 0, & \text{if } i = T(k,2) \text{ and } j = T(k,1) \\ A(i,j), & \text{otherwise.} \end{cases}$$

(Partition the set of triples, $\{(i,j,k) \mid i,j \in V, k = 1, \ldots, n-1\}$, into $\lceil n^3/\log n\rceil$ sets of at most $\lceil \log n\rceil$ elements each and assign one processor to each set. Then step 2 can be done in time $O(\log n)$ with $\lceil n^3/\log n\rceil$ processors.)

3. For $i = 1, \ldots, n-1$, let

$$f(i) = \begin{cases} 0, & \text{if } G_k \text{ is connected} \\ 1, & \text{otherwise.} \end{cases}$$

(By Corollary 1 of Theorem 5, this can be done in time $O(\log^2 n)$ with $n^2\lceil n/\log n\rceil$ processors.)

4. For $i = 1, \ldots, n-1$, let

$$y(i) = \begin{cases} i, & \text{if } f(i) = 1 \\ 0, & \text{otherwise.} \end{cases}$$

(constant time, $n-1$ processors)

5. Sort the array $y(1), \ldots, y(n-1)$ in decreasing order to obtain $z(1), \ldots, z(n-1)$. (As mentioned in Section 2.2, this can be done in time $O(\log n)$ with $n\lceil \log n\rceil$ processors.)

6. Let $m = \min \{i \mid i = 1, \ldots, n-1 \text{ and } z(i) = 0\}$. Let $K = m-1$. (The minimum of n numbers can be found in time $O(\log n)$ with $\lceil n/\log n\rceil$ processors.)

7. For $i = 1, \ldots, K$ and $j = 1,2$, let $B(i,j) = T(z(i),j)$. (constant time, n processors)

ALGORITHM BRIDGES (1) has time complexity $O(\log^2 n)$ and processor complexity $n^2\lceil n/\log n\rceil$. The best serial algorithm for finding the bridges of G uses an adjacency list representation of G and has time complexity

$O(m+n)$, where m is the number of edges of G [30]. If $m = O(n^2)$, this algorithm has time complexity $O(n^2)$. It follows from Theorem 14 of Section 8.2 that any algorithm which decides, from the adjacency matrix of G, whether or not G is bridge connected, must have time complexity at least cn^2, for some constant c. Comparing ALGORITHM BRIDGES (1) with the best serial algorithm shows a speedup of $O(n^2/\log^2 n)$ and an efficiency of $O(1/n\log n)$. It will be shown that the efficiency can be improved by using ALGORITHM BRIDGES (2) of Section 6.3.

6.2 Preliminary Algorithms

In this section, algorithms are presented to solve problems which arise in the improved algorithm, of Section 6.3, for finding the bridges of a graph.

Lemma 12. Let $V = \{1, \ldots, n\}$ and let $F : V \to V$ be a function. For each $i \in V$, let $F^0(i) = i$ and for $k = 1, \ldots, n-1$, let $F^k(i) = F(F^{k-1}(i))$. Then for a given $j \in V$, $F^k(j)$, for $k = 0, \ldots, n-1$ can be computed in time $O(\log n)$ with n processors.

Proof: For each $i \in V$, let $Y_0(i) = F(i)$ and for $t = 1, \ldots, \lceil \log(n-1) \rceil$, let $Y_t(i) = Y_{t-1}(Y_{t-1}(i))$. Then $Y_t(i) = F^{2^t}(i)$ for all $i \in V$. Assume that for $t = 0, \ldots, \lceil \log(n-1) \rceil$, $Y_t(i)$ has been computed for all $i \in V$. Assume that for some r, with $0 \le r \le \lceil \log(n-1) \rceil$, $F^k(j)$ has been computed for $0 \le k \le 2^r - 1$. Then $F^k(j)$, for $2^r \le k \le 2^{r+1}-1$, can be computed as follows. For $0 \le s \le 2^{r-1}$,

$$F^{2^r+s}(j) = F^{2^r}(F^s(j))$$

Since it was assumed that $F^s(j)$ has been computed and $F^{2^r}(i)$ has been computed for all $i \in V$, $F^{2^r}(F^s(j))$ can be computed in constant time. Thus the following steps compute $F^k(j)$, for $k = 0, \ldots, n-1$, from F.

1. Let $t = 0$ and let $Y_0(i) = F(i)$ for all $i \in V$.
2. If $t \geq \log(n-1)$, go to step 4, having computed $Y_0(i), \ldots, Y_{\lceil \log(n-1) \rceil}(i)$ for all $i \in V$. Otherwise, increment t by 1.
3. Let $Y_t(i) = Y_{t-1}(Y_{t-1}(i))$ for all $i \in V$.

Then return to step 2.

4. Let $r = 0$.
5. If $r \geq \log(n-1)$, stop, having computed $F^k(j)$ for $k = 0, \ldots, n-1$. Otherwise, increment r by 1.
6. For $s = 0, \ldots, 2^r - 1$, let $F^{2^r+s}(j) = F^{2^r}(F^s(j))$. Then return to step 5.

Each of steps 1 through 6 can be done in constant time with at most n processors. Since steps 2 and 3, and 5 and 6 are iterated at most $\lceil \log(n-1) \rceil + 1$ times, the entire algorithm can be done in time $O(\log n)$ with n processors. QED

<u>Theorem 8</u>. Let $G = (V,E)$ be a directed forest with n vertices. Let $F : V \to V$ be a function such that

$$\overrightarrow{E} = \{(i,F(i)) \mid i \in V, i \neq F(i)\}.$$

From F, the transitive closure of G can be found in time O(logn) with n^2 processors.

Proof: Let A be the adjacency matrix of G and let A^* be the transitive closure of A. For $i,j \in V$, $A^*(i,j) = 1$ if and only if $j = F^t(i)$ for some $t = 0, \ldots, n-1$. Thus A^* can be computed in the following way.

1. For each $i \in V$, compute $F^t(i)$ for $t = 0, \ldots, n-1$. (By Lemma 12, this can be done in time O(logn) with n^2 processors.)

2. For each $i \in V$, let

$$m(i) = \min \{t \mid t = 0, \ldots, n-1 \text{ and } F^t(i) = F^{t+1}(i)\}.$$

Then for each $i \in V$, $F^0(i), \ldots, F^{m(i)}(i)$ are distinct and $F^x(i) = F^y(i)$ for all $x,y \geq m(i)$.
(Since for each $i \in V$, the minimum of at most n numbers must be found, this can be done in time O(logn) with n[n/logn] processors.)

3. For each pair $i,j \in V$, let $A^*(i,j) = 0$. (constant time, n^2 processors.)

4. For each $i \in V$, replace $A^*(i,F^t(i))$ by 1 for $t = 0, \ldots, m(i)$. (constant time, n^2 processors)

Thus, computation of A^* from F can be carried out in time O(logn) with n^2 processors. Notice the necessity of step 2. If for some $i \in V$ there are x, y such that $x \neq y$ and $m(i) \leq x,y \leq n-1$, then in step 4, the two processors associated with i,x and i,y would both be instructed to store a 1 in the same memory location, $A^*(i,F^{m(i)}(i))$. This is not allowed by the model of computation. QED

<u>Corollary</u>. Let $G = (V,E)$ be a connected, undirected graph, where $V = \{1, . . ., n\}$. There is an algorithm which computes a function $F : V \to V$, so that the graph,

$$T = (V, \{\overrightarrow{(i,F(i))} \mid i \in V, i \neq F(i)\})$$

is a directed spanning tree of G, in time $O(\log^2 n)$ with n^2 processors, from the adjacency matrix of G.

<u>Proof:</u> This follows from Theorem 6 of Section 3.4 and Theorem 8. QED

Let $G = (V,E)$ be a rooted, directed tree and let $F : V \to V$ be a function such that

$$E = \{\overrightarrow{(i,F(i))} \mid i \in V, i \neq F(i)\}.$$

Let $b : V \to \{0,1\}$ be any function and let $f: V \to \{0,1\}$ be defined for $i \in V$ by

$$f(i) = \begin{cases} 1, & \text{if for some descendant } k \text{ of } i \text{ in } G, b(k) = 1 \\ 0, & \text{otherwise.} \end{cases}$$

In the improved bridges algorithm, it is necessary to compute f from F. Although this can be done in a straightforward way in time $O(\log^2 n)$ with n^2 processors, it will be shown in Theorem 9 that f can be computed with only $n\lceil \log n \rceil$ processors. This is the crucial step in reducing the number of processors used to find the bridges of a graph from n^3 to $n^2\lceil \log n \rceil$.

Define $F^k(i)$ for all $i \in V$ and for $k = 0, \ldots, n-1$ as before. For $i,j \in V$, j is a descendant of i if and only if $F^k(j) = i$ for some $k \in \{0, \ldots, n-1\}$. Let $Y_0(i) = F(i)$ for all $i \in V$. For $t = 1, \ldots, \lceil \log(n-1) \rceil$ let $Y_t(i) = Y_{t-1}(Y_{t-1}(i)) = F^{2^t}(i)$. Define $f_t : V \to \{0,1\}$, for all $i \in V$ and for $t = 0, \ldots, \lceil \log(n-1) \rceil$, by $f_t(i) = 1$ if and only if for some $v \in V$ and some k with $0 \le k < 2^t$, $F^k(v) = i$ and $b(v) = 1$. Then $f_{\lceil \log(n-1) \rceil}(i) = f(i)$ for all $i \in V$. Assume that for all i V and for $0 \le t \le \lceil \log(n-1) \rceil$, $Y_t(i)$ has been computed. It will be proven in Theorem 9 that f can be calculated from b in the following way.

ALGORITHM COLLECT

1. Let $t = 0$
2. For each $i \in V$,
   (a) let $f(i) = b(i)$
   (b) let $f_0(i) = f(i)$.
3. If $t \ge \log(n-1)$, stop, having computed $f_{\lceil \log(n-1) \rceil} = f$. Otherwise, increment t by 1.
4. For all $i \in V$, let

$$y_t(i) = \begin{cases} Y_{t-1}(i), \text{ if } f_{t-1}(i) = 1 \\ 0, \text{ otherwise.} \end{cases}$$

5. Sort $y_t(1), \ldots, y_t(n)$ in decreasing order to obtain $z_t(1), \ldots, z_t(n)$.

6. Let $s_t(1) = z_t(1)$ and for $i = 1, \ldots, n-1$, let

$$s_t(i+1) = \begin{cases} 0, & \text{if } z_t(i) = z_t(i+1) \\ z_t(i+1), & \text{otherwise.} \end{cases}$$

(Then all nonzero entries of the 1xn array s_t are distinct and every nonzero entry of the array y_t appears in the array s_t. Note that if the nonzero entries of s_t were not all distinct, then in step 7, two processors would attempt to change the content of the same memory location. This is not allowed by the model.)

7. For each $i \in V$ with $s_t(i) \neq 0$, replace $f(j)$ by 1, where $j = s_t(i)$. Then, for all $i \in V$, let $f_t(i) = f(i)$. Return to step 3.

(Notice that for all $j \in V$, if $f_t(j) = 1$, then $f_{t'}(j) = 1$ for all $t' \geq t$.)

In ALGORITHM COLLECT, $f_t(i) = 1$, for $t \geq 1$ if and only if $f_{t-1}(i) = 1$ or i appears in the array s_t. The vertex i appears in the array s_t if and only if there is a $v \in V$ such that $Y_{t-1}(v) = F^{2^{t-1}}(v) = i$ and $f_{t-1}(v) = 1$.

Lemma 13. For $t = 0, \ldots, \lceil \log(n-1) \rceil$ and $i \in V$, let $f_t(i)$ be as defined in ALGORITHM COLLECT. Then $f_t(i) = 1$ if and only if there is a vertex v of V with $b(v) = 1$ and $F^k(v) = i$, where $0 \leq k < 2^t$.

Proof: If $t = 0$, $f_0(i) = 1$ if and only if $b(i) = 1$ and therefore the lemma is true. Assume inductively that for some $t \geq 0$ the lemma is true. Assume $f_{t+1}(i) = 1$. If $f_t(i) = 1$, then by the induction hypothesis there is a vertex v with $b(v) = 1$ and $F^k(v) = i$, where $0 \leq k < 2^t < 2^{t+1}$, so the lemma is true. If $f_t(i) \neq 1$, then i must appear in the array s_{t+1}. Thus, there is a vertex w such that $f_t(w) = 1$ and $F^{2^t}(w) = i$. By the induction hypothesis, there is a vertex u and a k' such that $b(u) = 1$, $0 \leq k' < 2^t$, and $F^{k'}(u) = w$. But then $i = F^{2^t}(F^{k'}(u))$ and since $0 \leq 2^t \leq 2^t + k' < 2^{t+1}$, the lemma is true. Conversely, let i and v be vertices of V such that $b(v) = 1$ and $F^k(v) = i$, where $0 \leq k < 2^{t+1}$. If $f_t(i) = 1$, then $f_{t+1}(i) = 1$. Otherwise, by the induction hypothesis, $2^t < k < 2^{t+1}$. Let $k' = k - 2^t$. Then $0 < k' < 2^t$. Let $w = F^{k'}(v)$. By the induction hypothesis, $f_t(w) = 1$. Since

$$F^{2^t}(w) = F^{2^t}(F^{k-2^t}(v)) = i,$$

$Y_t(w) = i$, and thus i appears in the array s_{t+1}. Then $f_{t+1}(i) = 1$. QED

Thus, ALGORITHM COLLECT calculates $f_{\lceil \log(n-1)\rceil}$, where for $i \in V$, $f_{\lceil \log(n-1)\rceil}(i) = 1$ if and only if there is a $v \in V$ with $b(v) = 1$ and $F^k(v) = i$ for some k with $0 \leq k < 2^{\lceil \log(n-1)\rceil}$, that is, if and only if there is a descendant of i in G with $b(v) = 1$. Thus $f = f_{\lceil \log(n-1)\rceil}$.

Theorem 9. Let $G = (V,E)$ be a directed, rooted tree, with $V = \{1, \ldots, n\}$. Let $F : V \to V$ be a function such that

$$E = \{\overrightarrow{(i,F(i))} \mid i \in V, i \neq F(i)\}.$$

Let $b : V \to \{0,1\}$ be any function and define $f : V \to \{0,1\}$ for $i \in V$ by

$$f(i) = \begin{cases} 1, & \text{if for some descendant } j \text{ of } i, b(j) = 1 \\ 0, & \text{otherwise.} \end{cases}$$

Then f can be computed from F and b in time $O(\log^2 n)$ with $n\lceil \log n\rceil$ processors.

Proof: It was shown in the proof of Lemma 12 that for $t = 0, \ldots, \lceil \log(n-1)\rceil$, $Y_t(i) = F^{2^t}(i)$ can be computed from F for all $i \in V$ in time $O(\log n)$ with n processors. It has been shown in Lemma 13 that ALGORITHM COLLECT computes f, given b and the functions $Y_0, Y_1, \ldots, Y_{\lceil \log(n-1)\rceil}$. It remains to find the time and processor complexities of ALGORITHM COLLECT. All steps, except step 5 can be done in constant time with n processors. In step 5, n numbers must be sorted, which takes time $O(\log n)$ and $n\lceil \log n\rceil$ processors. Since steps 3 through 7 are iterated at most $\lceil \log(n-1)\rceil + 1$ times, ALGORITHM COLLECT can be done in time $O(\log^2 n)$ with $n\lceil \log n\rceil$ processors. Thus f can be computed from b and F in time

$$O(\log n) + O(\log^2 n) = O(\log^2 n)$$

with

$$\max \{n, n\lceil \log n\rceil\} = n\lceil \log n\rceil$$

processors. QED

6.3 Improved Bridge Algorithm

Let $G = (V,E)$ be a connected, undirected graph, where $V = \{1, . . ., n\}$. Let $T = (V,E(T))$ be a directed spanning tree of G and let $F : V \rightarrow V$ be such that

$$E(T) = \{\overrightarrow{(i,F(i))} \mid i \in V, i \neq F(i)\}.$$

Then the undirected graph $S = (V,E(S))$, where

$$E(S) = \{(i,F(i)) \mid i \in V \text{ and } F(i) \neq i\},$$

is a *spanning tree of G*. By Lemma 11, every bridge of G is in E(S). In the algorithm presented in this section for finding the bridges of G, a test is made on each vertex $i \in V$, with $i \neq F(i)$, to determine whether or not the edge $(i,F(i))$ of E(S) is a bridge of G. The test made on each vertex is based on the following theorem.

Theorem 10. Let i be a vertex of V such that $F(i) \neq i$. The edge $(i,F(i))$ of S is not a bridge of G if and only if either

(i) there is an edge of G joining a proper descendant of i and a nondescendant of i in S, or

(ii) there is an edge of G joining i and a nondescendant of i in S, other than F(i).

The proof of Theorem 10 relies on Lemma 14 below.

<u>Lemma 14</u>. The edge e of G is a bridge of G if and only if there is no cycle of G containing e.

<u>Proof</u>: See [9], p. 27. QED

<u>Proof of Theorem 10</u>: Assume that (i,F(i)) is not a bridge of G. Let $v_0, v_1, \ldots, v_k$ be a cycle in G containing (i,F(i)), where $v_0 = F(i) = v_k$ and $v_1 = i$. Then $k \geq 3$, so i,F(i) and v_2 are distinct vertices. If v_2 is a nondescendant of i in S, then the edge (i,v_2) of G is an edge as described in (ii) of the theorem. If v_2 is a descendant of i, let t be the largest index for which v_t is a proper descendant of i in S. Then $2 \leq t \leq k-1$ and v_{t+1} is a nondescendant of i in T and thus (v_t, v_{t+1}) is an edge of G as described in (i) of the theorem. The converse is proven as follows.

(i) Assume there are vertices j and k of V such that j is a proper descendant of i, k is a nondescendant of i and (j,k) is an edge of G. Let p be the simple path in S joining j and i. Let x be the youngest common ancestor of i and k in S. Let q be the simple path in S joining i and x, and r, the simple path in S joining k and x. Since i,j, and k are all distinct, $p \cup q \cup r \cup \{(j,k)\}$ is a cycle in G. Since k is a nondescendant of i, $x \neq i$, so that F(i) is a descendant of x. Then (i,F(i)) is on path q and is in the cycle $p \cup q \cup r \cup \{(j,k)\}$ and thus is not a bridge of G by Lemma 14.

(ii) Assume there is a vertex k of V such that k is a nondescendant of i, (i,k) is an edge of G, and $k \neq F(i)$. Let $x \in V$ be the youngest common ancestor of F(i) and k. Let p be the simple path in S joining F(i) and x, and q, the simple path joining k and x. Either p or q may contain no edges, in case $F(i) = x$ or $k = x$, but since $k \neq F(i)$, at least one of p and q must contain an edge. Then $p \cup q \cup \{(i,F(i)),(i,k)\}$ is a cycle of G containing (i,F(i)) so that (i,F(i)) is not a bridge of G, by Lemma 14. QED

ALGORITHM BRIDGES (2) will work in the following way. First, a function $F : V \to V$ is computed so that the graph

$$T = (V, \{\overrightarrow{(i,F(i))} \mid i \in V, F(i) \neq i\})$$

is a directed spanning tree of G. The adjacency matrix, M_T, of T is constructed and M_T^*, the transitive closure of M_T, is computed. Then for $i,j \in V$, j is a descendant of i in S, where

$$S = (V,\{(i,F(i)) \mid i \in V, i \neq F(i)\})$$

if and only if $M_T^*(i,j) = 1$. Next, for each i in V, a $1 \times n$ array, D_i, is computed, where for $j \in V$,

$$D_i(j) = \begin{cases} 1, & \text{if } j \text{ is a nondescendant of } i \text{ in } S \text{ and there is an edge} \\ & \text{of } G \text{ joining } j \text{ and a proper descendant of } i \text{ in } S \\ 1, & \text{if } j \text{ is a nondescendant of } i \text{ in } S \text{ and } j \neq F(i) \\ & \text{and } (i,j) \text{ is an edge of } G \\ 0, & \text{otherwise.} \end{cases}$$

AD-A056 888 ILLINOIS UNIV AT URBANA-CHAMPAIGN COORDINATED SCIENCE LAB F/G 9/2
PARALLEL ALGORITHMS FOR GRAPH THEORETIC PROBLEMS.(U)
AUG 77 C D SAVAGE DAAB07-72-C-0259
UNCLASSIFIED R-784 NL

2 OF 2
AD
A056888

END
DATE
FILMED
9-78
DDC

Then by Theorem 10, for each i in V, such that $i \neq F(i)$, the edge $(i,F(i))$ of G is a bridge of G if and only if the 1 x n array D_i does not contain a 1.

Calculation of D_i

For each $j \in V$, let $b_j : V \to \{0,1\}$ be defined for each $i \in V$ by

$$b_j(i) = \begin{cases} 1, & \text{if } (i,j) \text{ is an edge of } G,\ j \neq F(i), \text{ and } j \text{ is a nondescendant of } i \text{ in } S \\ 0, & \text{otherwise.} \end{cases}$$

For each $j \in V$, let $f^{(j)} : V \to V$ be defined for each $i \in V$ by

$$f^{(j)}(i) = \begin{cases} 1, & \text{if some descendant } k \text{ of } i \text{ has } b_j(k) = 1 \\ 0, & \text{otherwise.} \end{cases}$$

Lemma 15. Let the functions b_j and $f^{(j)}$, for $j \in V$, be defined as above. For each pair $i,j \in V$, let

$$D_i(j) = \begin{cases} 1, & \text{if } f^{(j)}(i) = 1 \text{ and } j \text{ is not a descendant of } i \text{ in } S \\ 0, & \text{otherwise.} \end{cases}$$

Then $D_i(j) = 1$ if and only if either

(i) j is a nondescendant of i and there is an edge of G joining j and a proper descendant of i or,

(ii) j is a nondescendant of i, $j \neq F(i)$, and (i,j) is an edge of G.

<u>Proof</u>: Assume $D_i(j) = 1$. Then j is a nondescendant of i and $f^{(j)}(i) = 1$. Thus, by definition of $f^{(j)}$, there is a descendant k of i for which $b_j(k) = 1$. If $k = i$, then by definition of b_j, (ii) holds. If $k \neq i$, then k is a proper descendant of i and thus (i) holds.

Conversely, assume j is a nondescendant of i in S.

(i) Assume there is a proper descendant k of i and an edge (k,j) of G. Then $j \neq F(k)$ and j is a nondescendant of k. Thus, by definition of b_j, $b_j(k) = 1$ so that $f^{(j)}(i) = 1$ and $D_i(j) = 1$.

(ii) If $j \neq F(i)$ and (i,j) is an edge of G then by definition of b_j, $b_j(i) = 1$. Thus $f^{(j)}(i) = 1$ and so $D_i(j) = 1$. QED

By Lemma 15, the following steps compute $D_i(j)$ for all $i,j \in V$, given A, the adjacency matrix of G.

1. Compute a function $F : V \to V$ so that the graph

$$T = (V, \{\overrightarrow{(i,F(i))} \mid i \in V, i \neq F(i)\})$$

is a directed spanning tree of G. Let M_T be the adjacency matrix of G. (By Theorem 6 and the corollary to Theorem 8, this can be done in time $O(\log^2 n)$ with n^2 processors.)

2. Compute M_T^*, the transitive closure of M_T. (By Theorem 8, this can be done in time $O(\log n)$ with n^2 processors.)

3. For each pair $i,j \in V$, let

$$b_j(i) = \begin{cases} 1, \text{ if } A(i,j) = 1,\ j \neq F(i) \text{ and } M_T^*(j,i) = 0 \\ 0, \text{ otherwise.} \end{cases}$$

(This can be done in constant time with n^2 processors.)

4. For each pair $i,j \in V$, compute $f^{(j)} : V \to \{0,1\}$ where $f^{(j)}(i) = 1$ if and only if some descendant k of j has $b_j(k) = 1$. (By Theorem 9, for a given $j \in V$, $f^{(j)}$ can be computed from b_j and F in time $O(\log^2 n)$ with $n\lceil \log n \rceil$ processors. Thus step 4 can be done in time $O(\log^2 n)$ with $n^2\lceil \log n \rceil$ processors.

5. For each pair $i,j \in V$, let

$$D_i(j) = \begin{cases} 1, \text{ if } f_j(i) = 1 \text{ and } M_T^*(j,i) = 0 \\ 0, \text{ otherwise.} \end{cases}$$

(constant time, n^2 processors)

Let K be the number of bridges of G. A $K \times 2$ array, B, where the set

$$\{(B(i,1),B(i,2)) \mid i = 1, \ldots, K\}$$

is the set of bridges of G, can be computed from the arrays $D_1, \ldots, D_n$ in the following way.

6. For each $i \in V$, let

$$m(i) = \begin{cases} 1, \text{ if } F(i) \neq i \text{ and the } 1 \times n \text{ array } D_i \text{ does not contain} \\ \quad \text{a } 1 \\ 0, \text{ otherwise.} \end{cases}$$

(It was shown in Section 2.2 that it is possible to determine whether or not an n-dimensional binary vector contains a 1 in time $O(\log n)$ with n processors. Thus, step 6 can be done in time $O(\log n)$ with n^2 processors.)

7. For each $i \in V$, let

$$y(i) = \begin{cases} i, & \text{if } m(i) = 1 \\ 0, & \text{otherwise.} \end{cases}$$

(constant time, n processors)

8. Sort the array $y(1), \ldots, y(n)$ in decreasing order to obtain the array $z(1), \ldots, z(n)$. ($O(\log n)$ time, $n\lceil \log n \rceil$ processors, as in Section 2.2)

9. Let $K' = \min \{j \in V \mid z(j) = 0\}$. Let $K = K' - 1$. ($O(\log n)$ time, $\lceil n/\log n \rceil$ processors, as in Section 2.2)

10. For $i = 1, \ldots, K$, let $B(i,1) = z(i)$ and $B(i,2) = F(z(i))$. (constant time, n processors)

After step 10, the set

$$\{(B(i,1),B(i,2)) \mid i = 1, \ldots, K\}$$

is the set of bridges of G. Note that G is bridge connected if and only if $K = 0$. The algorithm is summarized and analyzed below.

ALGORITHM BRIDGES (2)

Let $G = (V,E)$ be a connected, undirected graph, where $V = \{1, \ldots, n\}$. From A, the adjacency matrix of G, find

K, the number of bridges of G, and a K x 2 array, B, where

$$\{(B(i,1),B(i,2)) \mid i = 1, \ldots, K\}$$

is the set of bridges of G, by performing the steps 1 through 10 above.

Then ALGORITHM BRIDGES (2) has time complexity $O(\log^2 n)$ and processor complexity $n^2\lceil \log n \rceil$. As discussed in Section 6.1, the best serial algorithm for this problem has time complexity $O(n^2)$. The speedup in computation time which results from using the parallel algorithm instead of the serial algorithm is $O(n^2/\log^2 n)$ and the efficiency is $O(1/\log^3 n)$. Thus, a relatively small amount of waste is incurred by using $n^2\lceil \log n \rceil$ processors to achieve a speedup of $O(n^2/\log^2 n)$ in computing the bridges of a graph.

6.4 Bridge Connected Components

Let G = (V,E) be a connected, undirected graph. The bridge connected components of G can be computed using the following theorem.

Theorem 11. Let B be the set of bridges of G. A subgraph H of G is a bridge connected component of G if and only if it is a connected component of the graph $G' = (V,E-B)$.

Proof: Assume H is a connected component of G'. If (i,j) is a bridge of H, when (i,j) is removed from H there is no path in G' joining i and j. But since (i,j) is not a bridge of G, there must be a simple path $v_1, \ldots, v_k$ in G, where $i = v_1$, $j = v_k$ and for some t,

with $1 < t < k-1$, (v_t, v_{t+1}) is a bridge of G. This is a contradiction since $v_{t+1}, \ldots, v_k, v_1, \ldots, v_t$ is a path in G joining v_t and v_{t+1} which does not contain (v_t, v_{t+1}). Thus H is bridge connected. Let $K \neq H$ be a connected subgraph of G with $H \subseteq K$. There is a vertex v of K which is not in H. If u is a vertex of H, there is no path in G' joining u and v. Thus any path in K joining u and v must contain a bridge. Then K is not bridge connected and thus H is a bridge connected component of G.

Assume H is a bridge connected component of G. Then H is a connected subgraph of G'. Let K be the connected component of G' containing H. Since K is bridge connected and $H \subseteq K \subseteq G$, $H = K$ and thus H is a connected component of G'. QED

By Theorem 11, the bridge connected components of G can be computed by finding the bridges of G, removing all bridges from G to obtain a graph G', and then finding the connected components of G'. The algorithm below uses this technique to compute the bridge connected components of G in time $O(\log^2 n)$ with $n^2 \lceil \log n \rceil$ processors.

ALGORITHM BRIDGE CONNECTED COMPONENTS

Let $G = (V,E)$ be a connected, undirected graph, where $V = \{1, \ldots, n\}$. From the adjacency matrix, A, of G, compute a $1 \times n$ array D where $D(i) = D(j)$ if and only if i and j are in the same bridge connected component of G.

1. Compute a $K \times 2$ array B, where K is the number of bridges of G and the set

$$\{(B(i,1),B(i,2)) \mid i = 1, . . ., K\}$$

is the set of bridges of G. (This can be done in time $O(\log^2 n)$ with $n^2\lceil \log n \rceil$ processors by using ALGORITHM BRIDGES (2).)

2. For i = 1, . . ., K, replace A(B(i,1),B(i,2)) by 0. A is now the adjacency matrix of G'. (constant time, $K \le n$ processors)

3. Compute a 1 x n array, D, where D(i) = D(j) if and only if i and j are in the same connected component of G'. (By Theorem 5, this can be done in time $O(\log^2 n)$ with $n\lceil n/\log n \rceil$ processors.)

Chapter 7

CYCLES

In this chapter, problems involving cycles in a graph are investigated. In Section 7.1, it is shown that if G is a connected, undirected graph with n vertices, a cycle of G can be found in time $O(\log^2 n)$ with n^2 processors, if one exists, and a cycle basis for G can be found in time $O(\log^2 n)$ with n^3 processors. In Section 7.2, an algorithm is presented which finds a cycle of minimum length, if one exists, in a directed graph with n vertices in time $O(\log^2 n)$ with n^3 processors. For the problem of finding a cycle of a directed graph G, no parallel algorithm has been found which is more efficient than finding the shortest cycle of G.

7.1 Cycles in an Undirected Graph

Finding a Cycle in an Undirected Graph

Let $G = (V,E)$ be a connected, undirected graph, where $V = \{1, \ldots, n\}$, and let S be a spanning tree of G. If there is an edge e of G which is not an edge of S, adding e to S creates a cycle in S. The algorithm presented here for finding a cycle of G will first find a spanning tree, S, of G and vertices x and y of V such that $(x,y) \in E$, but (x,y) is not an edge of S. Next, the cycle $C_{(x,y)}$, created by adding (x,y) to S, is computed. If yca(x,y) is the youngest common ancestor of x and y in S, then $C_{(x,y)}$ consists of the edge (x,y) and the simple paths in S joining yca(x,y) with x and y.

Let $F : V \to V$ be a function such that the directed graph

$$T = (V, \overrightarrow{\{(i,F(i)) \mid i \in V, F(i) \neq i\}})$$

is a directed spanning tree of G. Then the undirected graph

$$S = (V, \{(i,F(i)) \mid i \in V, i \neq F(i)\})$$

is a spanning tree of G. If x and y are vertices of V, the youngest common ancestor of x and y in S can be found in the following way.

(a) Construct the adjacency matrix M_T of T.

(b) Find M_T^*, the transitive closure of M_T. Then for $i,j \in V$, j is a descendant of i in S if and only if $M_T^*(j,i) = 1$.

(c) Define a relation R on V by jRi if and only if $M_T^*(j,i) = 1$. Then

$$yca(x,y) = \min_R \{i \in V \mid xRi \text{ and } yRi\}.$$

<u>Lemma 16</u>. Let $S = (V,E')$ be a tree and $F : V \to V$ a function such that

$$E' = \{(i,F(i)) \mid i \in V, i \neq F(i)\}.$$

Then for a given pair x,y of V, yca(x,y) can be computed in time O(logn) with n^2 processors.

<u>Proof</u>: Steps (a) through (c) above compute yca(x,y) from F. Step (a) can be done in constant time with n^2 processors, step (b), in time O(logn) with n^2 processors, by Theorem 8, and step (c), in time O(logn) with $\lceil n/\log n \rceil$ processors, as in Section 2.2. Thus yca(x,y) can be computed in time O(logn) with n^2 processors. QED

<u>Corollary</u>. Let S and F be as in Lemma 16. Then yca(i,j) can be computed for all $i,j \in V$ in time O(logn) with $n^2\lceil n/\log n \rceil$ processors.

<u>Proof</u>: In step (c), preceding Lemma 16, for <u>all</u> $i,j \in V$ let

$$yca(i,j) = \min_R \{v \in V \mid iRv \text{ and } jRv\}.$$

Then steps (a) through (c) compute $yca(i,j)$ for all $i,j \in V$. Since the new step (c) can be done in time $O(\log n)$ with $n^2\lceil n/\log n\rceil$ processors, the corollary follows. QED

To find vertices x and y of V such that $(x,y) \in E$, but (x,y) is not an edge of $S = (V,E')$, define $g : V \to V$, for $i \in V$, by

$$g(i) = \begin{cases} 0, & \text{if no edge in } E - E' \text{ is incident with } i \\ \min\{j \in V \mid (i,j) \in E - E'\}, & \text{otherwise.} \end{cases}$$

If $g(i) = 0$ for all $i \in V$, then G contains no cycles. Otherwise, let $h = \min\{i \in V \mid g(i) \neq 0\}$. Then $(h,g(h))$ is an edge of G which is not in S. Let $w = yca(h,g(h))$. Let r and s be the smallest integers such that $F^r(h) = w$ and $F^s(g(h)) = w$. Then the cycle

$$h, F(h), \ldots, F^r(h), F^{s-1}(g(h)), \ldots, F(g(h)), g(h), h$$

is the cycle of G created by adding $(h,g(h))$ to S.

Define $C_{(h,g(h))} : V \to V$, for $i \in V$, by

$$C_{(h,g(h))}(i) = \begin{cases} F(i), & \text{if } i = F^k(h) \text{ and } 0 \leq k \leq r-1 \\ F^{k-1}(g(h)), & \text{if } i = F^k(g(h)) \text{ and } 1 \leq k \leq s \\ 0, & \text{otherwise.} \end{cases}$$

Then the sequence

$$C^0_{(h,g(h))}(h),\ C^1_{(h,g(h))}(h),\ .\ .\ .\ ,\ C^{n-1}_{(h,g(h))}(h)$$

consists of the vertices of the cycle created by adding (h,g(h)) to S, in the order in which they occur, followed by a sequence of zeroes. The entire algorithm is presented and analyzed below.

ALGORITHM CYCLE

Let G = (V,E) be a connected, undirected graph, where V = {1, . . ., n}. From the adjacency matrix, A, of G, find a cycle of G in the following way.

1. *Find a function* F : V→V *so that the graph*

$$T = (V, \overrightarrow{\{(i,F(i)} \mid i \in V, i \neq F(i)\})$$

is a directed spanning tree of G and let M_T be the adjacency matrix of T. (By Theorem 6 and the corollary to Theorem 8, this can be done in time $O(\log^2 n)$ with n^2 processors.)

2. Compute M_T^*. (By Theorem 8, this can be done in time O(logn) with n^2 processors.)

3. For each i ε V with F(i) ≠ i, replace A(i,F(i)) and A(F(i),i) by 0. (constant time, n processors)

4. For each i ε V, let

$$g(i) = \begin{cases} 0, & \text{if the vector } [A(i,1), \ldots, A(i,n)] \text{ does not contain a 1} \\ \min\{j \in V \mid A(i,j) = 1\}, & \text{otherwise.} \end{cases}$$

(By the results of Section 2.2, step 4 can be done in time O(logn) with n^2 processors.)

5. If the vector [g(1), . . ., g(n)] has no nonzero entry, stop, and indicate that G has no cycles. Otherwise, let $h = \min\{i \in V \mid g(i) \neq 0\}$. (It was shown in Section 2.2 that this can be done in time O(logn) with n processors.)

6. Find yca(h,g(h)). (By Lemma 16, this can be done in time O(logn) with n^2 processors.)

7. For all $i \in V$, compute $C_{(h,g(h))}$ as follows. First, let $C_{(h,g(h))}(i) = 0$ for all $i \in V$. Then,

(a) if $i \neq yca(h,g(h))$ and $M_T^*(h,i) = 1$ and $M_T^*(i,yca(h,g(h))) = 1$, replace $C_{(h,g(h))}(i)$ by F(i).

(b) if $i \neq yca(h,g(h))$ and $M_T^*(g(h),i) = 1$ and $M_T^*(i,yca(h,g(h))) = 1$, replace $C_{(h,g(h))}(F(i))$ by i.

(Step 7 can be done in constant time with n processors.

8. Compute $C^k_{(h,g(h))}(h)$ for k = 0, . . ., n-1. (By Lemma 12, this can be done in time O(logn) with n processors.)

Thus, ALGORITHM CYCLE can be performed in time $O(\log^2 n)$ with n^2 processors. The best serial algorithm for finding a cycle in an undirected graph uses an adjacency list representation of the graph and has a worst case time complexity of $O(n^2)$ [21]. Thus, ALGORITHM CYCLE achieves a speedup of $O(n^2/\log^2 n)$ over the best serial algorithm with an efficiency of $O(1/\log^2 n)$.

Finding a Cycle Basis

Let $G = (V,E)$ be an undirected graph. Let $H = (V_H,E_H)$ and $K = (V_K,E_K)$ be subgraphs of G. The symmetric difference of H and K, written $H \oplus K$, is the subgraph $G' = (V',E')$ of G where

$$E' = \{e \mid e \in E_H \cup E_K \text{ and } e \notin E_H \cap E_K\}$$

and

$$V' = \{v \in V \mid v \text{ is incident with some edge of } E'\}.$$

A set of fundamental cycles of G is a collection, C, of cycles of G with the property that any cycle C of G can be written as $C = C_1 \oplus \ldots \oplus C_k$ for some cycles $C_1, \ldots, C_k \in C$ [21]. A cycle basis for G is a minimal collection of fundamental cycles of G. It can be shown that if G has t connected components, any cycle basis for G has $m - n + t$ elements, where m is the number of edges of G and n, the number of vertices ([4], p. 25). Let S be a spanning forest for G. Let $\{e_1, \ldots, e_{m-n+t}\}$ be the edges of G which are not in S. Let C_i be the cycle of G created by adding edge e_i to S. Then $C = \{C_1, \ldots, C_{m-n+t}\}$ is a set of fundamental cycles for G (see [21]) of cardinality m-n+t and thus C is a cycle basis for G.

Let $G = (V,E)$ be a connected, undirected graph with $V = \{1, \ldots, n\}$. ALGORITHM CYCLE BASIS below finds a cycle basis for G in the following way. First, a spanning tree $S = (V,E')$ of G is found. Then, for each $x,y \in V$ such that $(x,y) \in E$ and $(x,y) \notin E'$, the cycle $C_{(x,y)}$, obtained by adding edge (x,y) to S is computed as in ALGORITHM CYCLE.

ALGORITHM CYCLE BASIS

Let $G = (V,E)$ be a connected, undirected graph, where $V = \{1, . . . , n\}$. Find a cycle basis of G in the following way.

1. Find a function $F : V \to V$ such that the graph

$$T = (V, \{\overrightarrow{(i,F(i))} \mid i \in V, i \neq F(i)\})$$

is a directed spanning tree of G. (By the corollary to Theorem 8, this can be done in time $O(\log^2 n)$ with n^2 processors.)

2. Find yca(i,j) for all $i,j \in V$. (By the corollary to Lemma 16, this can be done in time O(logn) with $n^2\lceil n/\log n\rceil$ processors.)

3. For each $i \in V$ with $F(i) \neq i$, replace $A(i,F(i))$ and $A(F(i),i)$ by 0. Then $A(x,y) = 1$ if and only if $(x,y) \in E - E'$. (constant time, n processors)

4. For each $x,y \in V$ with $A(x,y) = 1$, find $C^k_{(x,y)}(x)$ for $k = 0, . . . , n-1$ using steps 7 and 8 of ALGORITHM CYCLE. (Since steps 7 and 8 of ALGORITHM CYCLE can be done in time O(logn) with n processors, step 4 can be done in time O(logn) with n^3 processors.)

If it is desired, a procedure like the one which stored K biconnected components in a K x n array can be used to store the cycles in the set

$$\{C_{(x,y)} \mid (x,y) \in E - E'\}$$

in an (m-n+1) x n array.

ALGORITHM CYCLE BASIS has time complexity $O(\log^2 n)$ and processor complexity n^3. The best serial algorithm for this problem uses an adjacency list representation of the graph and has time complexity $O(n^3)$ [21]. The parallel algorithm for finding a cycle basis achieves a speedup of $O(n^3/\log^2 n)$ over the serial algorithm, with an efficiency of $O(1/\log^2 n)$. Thus a very large speedup is achieved by the parallel algorithm with relatively little waste.

7.2 Shortest Cycles

Let $G = (V,E)$ be a simple, directed graph with $V = \{1, \ldots, n\}$. The algorithm of this section finds a cycle of minimum length in G in time $O(\log^2 n)$ with $n^2\lceil n/\log n\rceil$ processors. It is based on knowledge of two matrices associated with G. The <u>shortest path matrix</u> of G is an n x n matrix P where for $i,j \in V$,

$$P(i,j) = \begin{cases} \infty, & \text{if there is no path in G from i to j} \\ d(i,j), & \text{the length of the shortest path in G from i to j, otherwise.} \end{cases}$$

A <u>next vertex matrix</u> of G is an n x n matrix N, where for $i,j \in V$,

$$N(i,j) = \begin{cases} 0, & \text{if there is no path in G from i to j} \\ i, & \text{if } i = j \\ k, & \text{where the edge } \overrightarrow{(i,k)} \text{ is on a shortest path in G from i to j, otherwise.} \end{cases}$$

Notice that if $N(i,j) \neq 0$, then the path

$$i,\ N(i,j),\ N(N(i,j),j),\ \ldots,\ j$$

is a shortest path in G from i to j. It will be shown in Lemma 17 that P and N can be calculated from the adjacency matrix, A, of G in time $O(\log^2 n)$ with $n^2\lceil n/\log n\rceil$ processors by the following steps.

<u>Calculation of P and N from A</u>

1. For all $i \in V$, let

$$P_0(i,j) = \begin{cases} 1, & \text{if } A(i,j) = 1 \\ 0, & \text{if } i = j \\ \infty, & \text{otherwise} \end{cases}$$

$$N_0(i,j) = \begin{cases} i, & \text{if } i = j \\ j, & \text{if } A(i,j) = 1 \\ 0, & \text{otherwise} \end{cases}$$

$$m_0(i,j) = \begin{cases} i, & \text{if } i = j \\ j, & \text{if } A(i,j) = 1 \\ 0, & \text{otherwise} \end{cases}$$

(constant time, n^2 processors)

2. Let $t = 0$. (constant time, 1 processor)

3. If $t \geq \log(n-1)$, stop, and let $P = P_{\lceil\log(n-1)\rceil}$ and $N = N_{\lceil\log(n-1)\rceil}$. Otherwise, increment t by 1. (constant time, n^2 processors)

4. For all $i,j \in V$, let

$$P_t(i,j) = \min\{P_{t-1}(i,k) + P_{t-1}(k,j) \mid k \in V\}.$$

(Since the minimum of n numbers can be found in time $O(\log n)$ with $\lceil n/\log n \rceil$ processors, step 4 can be done in time $O(\log n)$ with $n^2 \lceil n/\log n \rceil$ processors.)

5. For each $i,j \in V$, let

$$m_t(i,j) = \begin{cases} i, \text{ if } P_t(i,j) = 0 \\ 0, \text{ if } P_t(i,j) = \infty \\ \min\{k \in V \mid k \neq i \text{ and } P_t(i,j) = P_{t-1}(i,k) + P_{t-1}(k,j)\}, \\ \quad \text{otherwise.} \end{cases}$$

Note that if $0 < P_t(i,j) < \infty$, then there is a $k \in V$, $k \neq i$, such that $P_t(i,j) = P_{t-1}(i,k) + P_{t-1}(k,j)$. As in step 4, this can be done in time $O(\log n)$ with $n^2 \lceil n/\log n \rceil$ processors.)

6. For each $i,j \in V$, let

$$N_t(i,j) = \begin{cases} 0, \text{ if } m_t(i,j) = 0 \\ N_{t-1}(i, m_t(i,j)), \text{ otherwise.} \end{cases}$$

Then return to step 3. (constant time, n^2 processors)

<u>Lemma 17</u>. Let $G = (V,E)$ be a simple, directed graph, with $V = \{1, \ldots, n\}$, and let P be the shortest path matrix of G, and N the next vertex matrix of G. Then P and N can be computed from A, the adjacency matrix of G, in time $O(\log^2 n)$ with $n^2 \lceil n/\log n \rceil$ processors.

<u>Proof</u>: It will be shown first that steps 1 through 6 above compute P and N.

<u>Claim</u>: For $t = 0, \ldots, \lceil \log(n-1) \rceil$ and $i,j \in V$,

$$\text{(a)} \quad P_t(i,j) = \begin{cases} P(i,j), & \text{if there is a path in G from i to j of length at most } 2^t \\ \infty, & \text{otherwise} \end{cases}$$

$$\text{(b)} \quad N_t(i,j) = \begin{cases} i, & \text{if } P_t(i,j) = 0 \\ 0, & \text{if } P_t(i,j) = \infty \\ k, & \text{where the edge } \overrightarrow{(i,k)} \text{ of G is on a shortest path in G from i to j, otherwise.} \end{cases}$$

The claim will be proven by induction on t. If $t = 0$, the claim is true. Assume that for some $t \geq 0$, the claim is true.

(a) If there is no path from i to j in G of length at most 2^{t+1}, then by the induction hypothesis, for all $k \in V$, either $P_t(i,k) = \infty$ or $P_t(k,j) = \infty$, so that $P_{t+1}(i,j) = \infty$. Otherwise, let q be a shortest path from i to j in G of length at most 2^{t+1}. Then there is a vertex k of q such that $P(i,k) \leq 2^t$, $P(k,j) \leq 2^t$ and $P(i,j) = P(i,k) + P(k,j)$. Thus, by the induction hypothesis,

$$P(i,j) = P_t(i,k) + P_t(k,j) = P_{t+1}(i,j).$$

(b) If $P_{t+1}(i,j) = 0$, then $m_{t+1}(i) = i$ and $N_{t+1}(i,j) = N_t(i,i) = i$. If $P_{t+1}(i,j) = \infty$, then $m_{t+1}(i) = 0$, thus $N_{t+1}(i,j) = 0$. If $0 < P_{t+1}(i,j) < \infty$, then

$$N_{t+1}(i,j) = N_t(i,m_{t+1}(i,j)).$$

By the induction hypothesis, $N_t(i,m_{t+1}(i,j))$ is a vertex $k \neq i$, such that the edge $\overrightarrow{(i,k)}$ is on a shortest path in G from i to $m_{t+1}(i,j)$. Since $m_{t+1}(i,j)$ is a vertex distinct from i on a shortest path in G from i to j, $N_{t+1}(i,j) = k$, and $\overrightarrow{(i,k)}$ is an edge on a shortest path in G from i to j. This completes the proof of the claim.

Since the length of the shortest path in G from i to j, for any $i,j \in V$, is at most n-1, it follows from the claim that $P_{\lceil \log(n-1) \rceil} = P$ and $N_{\lceil \log(n-1) \rceil} = N$. Thus, the steps 1 through 6 preceding the lemma compute P and N. It remains to find the time and processor complexities of steps 1 through 6. Steps 1 and 2 can be done in constant time with n^2 processors. Steps 3 through 6 are iterated at most $\lceil \log(n-1) \rceil + 1$ times and one iteration can be done in time $O(\log n)$ with $n^2 \lceil n/\log n \rceil$ processors. Thus, P and N can be calculated in time $O(\log^2 n)$ with $n^2 \lceil n/\log n \rceil$ processors. QED

The computation of a shortest cycle of G from the matrices P and N is based on the following lemma.

<u>Lemma 18</u>. If $C = (V_C, E_C)$ is a shortest cycle of a directed graph G, where $V_C = \{v_1, \ldots, v_K\}$, and

$$E_C = \{(v_i, v_{i+1}) \mid i = 1, \ldots, k-1\} \quad \{(v_k, v_1)\},$$

then the path q from v_1 to v_k in C is a shortest path from v_1 to v_k in G.

<u>Proof</u>: Let p be a simple path in G from v_1 to v_k of length less than the length of q. Then $p \cup \{(v_k, v_1)\}$ is a cycle of G, shorter than C. This contradicts the definition of C. QED

By Lemma 18, a shortest cycle in G may be computed by finding vertices $x,y \in V$ such that

(i) $\overrightarrow{(y,x)}$ is an edge of G and

(ii) $P(x,y) = \min\{P(i,j) \mid i,j \in V \text{ and } \overrightarrow{(j,i)} \text{ is an edge of } G\}$.

Then a shortest cycle in G consists of the union of the edge $\overrightarrow{(y,x)}$ and a shortest path in G from x to y (which can be calculated from N). These steps are made precise in ALGORITHM SHORT CYCLE (D) below.

ALGORITHM SHORT CYCLE (D)

Let $G = (V,E)$ be a simple, directed graph with $V = \{1, \ldots, n\}$. Find a cycle of G of minimum length in the following way from the adjacency matrix, A, of G.

1. Compute P, the shortest path matrix and N, the next vertex matrix of G. (By Lemma 17, this can be done in time $O(\log^2 n)$ with $n^2\lceil n/\log n\rceil$ processors.)

2. Let

$$m = \min\{P(i,j) \mid i,j \in V,\ A(j,i) = 1\}.$$

If $m = \infty$, stop, and indicate that G has no cycles. (The minimum of at most n^2 numbers can be found in time $O(\log n)$ with $\lceil n^2/\log n\rceil$ processors.)

3. For all $i \in V$, let $\vec{v}_i(j)$ be defined for each $j \in V$ by

$$\vec{v}_i(j) = \begin{cases} 1, & \text{if } A(i,j) = 1 \text{ and } P(i,j) = m \\ 0, & \text{otherwise.} \end{cases}$$

(constant time, n^2 processors)

4. For all $i \in V$, let

$$f(i) = \begin{cases} 1, & \text{if the n dimensional vector } \vec{v}_i \text{ contains a 1} \\ 0, & \text{otherwise} \end{cases}$$

(This involves determining, for each $i \in V$, whether a binary vector contains a 1. Thus, step 3 can be done in time $O(\log n)$ with n^2 processors.)

5. Let $x = \min \{i \in V \mid f(i) \neq 0\}$. (Note that if $m \neq \infty$, such x and y exist.) Let

$$y = \min \{j \in V \mid A(j,x) = 1 \text{ and } P(x,j) = m\}.$$

($O(\log n)$ time, $\lceil n/\log n \rceil$ processors, as in Section 2.2)

6. For all $i \in V$, let

$$C(i) = \begin{cases} N(i,y), & \text{if } i \neq y \\ 0, & \text{otherwise.} \end{cases}$$

(constant time, n processors)

7. Compute $C^t(x)$ for $t = 0, \ldots, m$. Then the cycle

$$x, C^1(x), \ldots, C^m(x) = y, x$$

is a cycle of G of minimum length. (By Lemma 12, this can be done in time $O(\log n)$ with n processors.)

ALGORITHM SHORT CYCLE (D) has time complexity $O(\log^2 n)$ and processor complexity $n^2\lceil n/\log n\rceil$. Notice that this algorithm solves the problem of finding a cycle of G. The best known serial algorithms for detecting a cycle in a directed graph has a worst case time complexity of $O(n^2)$ [15,21]. Thus the parallel algorithm achieves a speedup of $O(n^2/\log^2 n)$ with an efficiency of $O(1/n\log n)$.

Chapter 8

LOWER BOUNDS

In previous chapters, algorithms of time complexity $O(\log^2 n)$ were found for several graph problems. It will now be shown that at least $2 \log n + c$ time units are required to solve some of these problems on the unbounded parallel model, when the adjacency or weight matrix of the graph is used as input. Thus in many cases, the algorithms obtained in this paper are within a factor of log n of being optimal, as far as time complexity is concerned.

The lower bounds are obtained in two steps. First, an argument is presented to show that the solution to a given problem on a graph with n vertices is a nontrivial function of at least $g(n)$ entries in the adjacency matrix of the graph, for some function g. This argument, whenever possible, is based on the Aanderaa-Rosenberg Theorem of Section 8.2. Second, Theorem 12 of Section 8.1 is used to show that at least $\lceil \log(g(n)) \rceil$ time units are required to compute a quantity which is a function of $g(n)$ input variables. In each problem considered, $g(n)$ will be such that $\lceil \log(g(n)) \rceil \geq 2 \log n + c$, for some constant c.

8.1 A Fan-In Theorem

A <u>binary tree</u>, T, is a rooted tree in which each vertex has at most two sons. A vertex of T which has no sons is called a <u>leaf</u>. The <u>height</u> of T is the length of the longest simple path in T between the

root of T and a leaf of T. Note that a binary tree of height h can have at most 2^h leaves.

Let D be a set, and f a function defined on D^n for some positive integer n. Let A be an algorithm which computes the function f, given $(x_1, \ldots, x_n) \in D^n$, on the unbounded parallel model in t time units. In computing f, A may compute several intermediate functions. The computation of f by A can be represented by a <u>binary computation tree</u>, T_A, with height t, which is constructed in the following way. At level t, place a single vertex, the root of T_A, labeled f. Assume inductively that T_A has been constructed through level i where $0 < i \le t$ and each vertex at level i is labeled by a function of $(x_1, \ldots, x_n)$ which has been computed by algorithm A in at most i time units. Level i - 1 of T_A is constructed as follows. If a vertex v at level i is labeled g, where $g(x_1, \ldots, x_n) \in \{x_1, \ldots, x_n\}$, then v is a leaf of T_A. Otherwise, there must be functions $h(x_1, \ldots, x_n)$ and $h'(x_1, \ldots, x_n)$ computed by A in at most i - 1 time units, and a binary function F, such that

$$g(x_1, \ldots, x_n) = F(h(x_1, \ldots, x_n), h'(x_1, \ldots, x_n)).$$

In this case, two vertices labeled h and h' are added to level i - 1 of T_A, along with the edges (g,h) and (g,h').

As an example, let $f(x_1, \ldots, x_5) = \sum_{i=1}^{5} x_i$. Then f is computed by the algorithm below.

Algorithm A

Input: $x_1, \ldots, x_5$

1. Assign one processor to the pair (x_1,x_2) and one processor to the pair (x_4,x_5) to compute $x_1 + x_2$ and $x_4 + x_5$.
2. Assign one processor to the pair (x_1+x_2,x_3) to compute $(x_1+x_2) + x_3$.
3. Assign one processor to the pair $((x_1+x_2) + x_3, x_4+x_5)$ to compute $((x_1+x_2) + x_3) + (x_4+x_5) = f(x_1, \ldots, x_5)$.

Algorithm A computes f in 3 time units. A binary computation tree for A is shown in Figure 5.

For $i \in \{1, \ldots n\}$, $f(x_1, \ldots, x_n)$ is a nontrivial function of x_i if there are elements $(a_1, \ldots, a_n)$ and $(b_1, \ldots, b_n)$ of D^n such that $a_i \neq b_i$, but $a_j = b_j$ for $j \neq i$, and $f(a_1, \ldots, a_n) \neq f(b_1, \ldots, b_n)$.

Theorem 12. Let $f(x_1, \ldots, x_n)$ be a nontrivial function of each of the variables $x_1, \ldots, x_n$. Then any algorithm which computes f, given $x_1, \ldots, x_n$, on the unbounded parallel model, has time complexity at least $\lceil \log n \rceil$.

Proof: Let A be any algorithm which computes f and let T_A be the binary computation tree for A. Since f is a nontrivial function of x_i, for $i = 1, \ldots, n$, there must be a path in T_A from the root, f, to a leaf labeled x_i. Thus, T_A must have at least n leaves. Since a binary tree of height h has at most 2^h leaves, the height of T_A is at least $\log n$, and thus at least $\lceil \log n \rceil$, since the height is an integer. Thus, A requires at least $\lceil \log n \rceil$ time units to compute f. QED

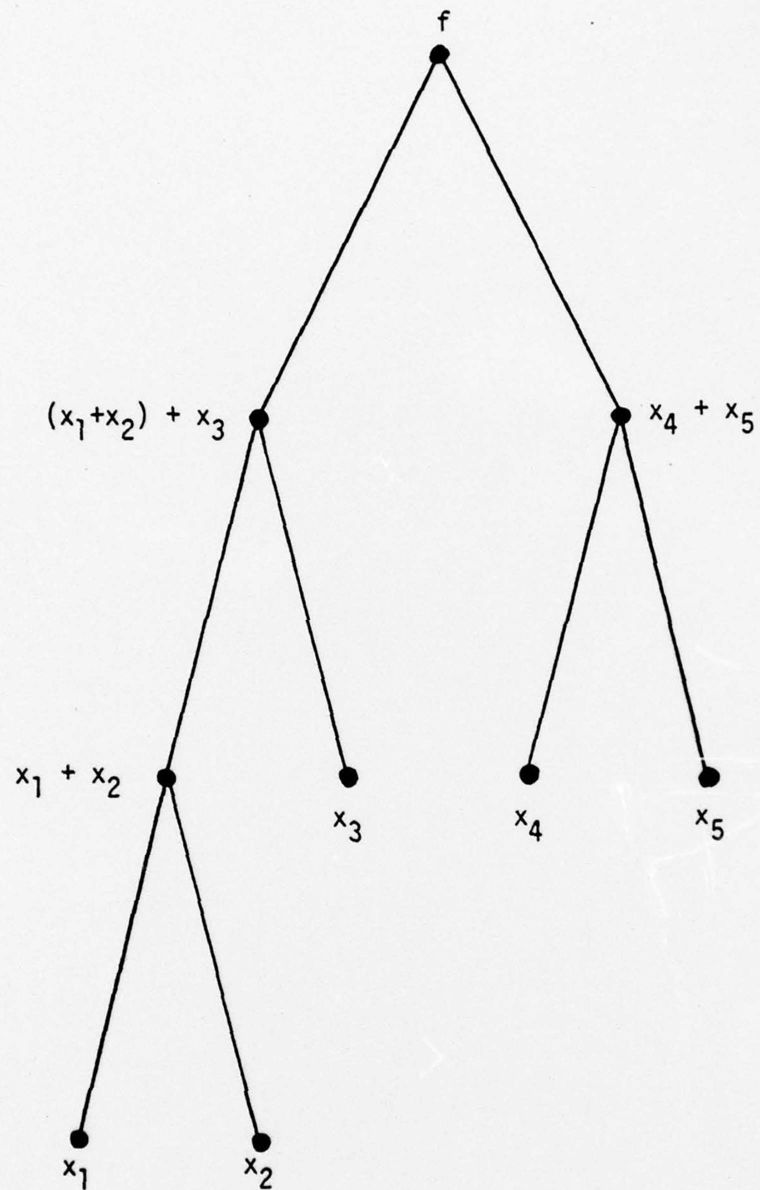


Figure 5. A binary computation tree for Algorithm A

8.2 The Aanderaa-Rosenberg Theorem

Let P_n *be a property of undirected graphs with n vertices.* For example, P_n may be the property "being connected" or "being planar." P_n is called nontrivial if there is some graph with n vertices which has property P_n and some graph with n vertices which does not have property P_n. "Being planar" is a nontrivial property of graphs with $n \geq 5$ vertices. P_n is monotone if whenever a graph $G = (V,E)$ with n vertices has property P_n, then any graph $G' = (V,E')$, where $E \subseteq E'$, also has property P_n. P_n is symmetric under vertex permutation if whenever a graph $G = (V,E)$ has property P_n, and $p : V \to V$ is a permutation of V, then the graph $G' = (V,E')$ also has property P_n, where

$$E' = \{(p(x),p(y)) \mid (x,y) \in E\}.$$

The following theorem was conjectured by Aanderaa and Rosenberg [23] and has been proven by Rivest and Vuillemin [22].

Theorem 13. Let P_n be a nontrivial, monotone property of undirected graphs with n vertices, which is symmetric under vertex permutation. For an undirected graph G, with n vertices, let f_{P_n} be a function of the entries of the adjacency matrix of G which has the value 1, if G has property P_n, and has value 0, otherwise. Then f_{P_n} is a nontrivial function of at least cn^2 entries of the adjacency matrix of G, for some constant c.

The following lemma is easily verified.

Lemma 19. The following properties of undirected graphs with n vertices are nontrivial, monotone, and symmetric under vertex permutation:

(i) being biconnected, for $n \geq 2$

(ii) being bridge connected, for $n \geq 2$, and

(iii) possessing a cycle, for $n \geq 3$.

8.3 Lower Bound Results

Using the results of the previous two sections, the following lower bounds are obtained.

Theorem 14. On the unbounded parallel model, at least $2 \log n + c$ time units, for some constant c, are required to determine

(i) if a given undirected graph G, with $n \geq 2$ vertices, is biconnected,

(ii) if a given undirected graph G, with $n \geq 2$ vertices, is bridge connected, and

(iii) if a given undirected graph G, with $n \geq 3$ vertices, contains a cycle,

if the adjacency matrix of G is used as the input.

Proof: (i) Let f have the value 1, if G is biconnected, 0, otherwise. Any algorithm which determines if G is biconnected, using the adjacency matrix of G as input, computes f as a function of the entries in the adjacency matrix. By Lemma 19 (i) and Theorem 13, f is a nontrivial function of at least kn^2 input variables for some constant k. Thus, by Theorem 12, at least

$$\lceil \log(kn^2) \rceil \geq 2 \log n + \log k$$

time units are required to compute f.

The proofs of (ii) and (iii) are analogous. QED

To obtain lower bounds on the time required to determine if a directed graph contains a cycle, to find the dominators of a rooted, directed graph, and to find a minimum spanning tree of a weighted graph, different techniques must be used. The first lemma below is due to Holt and Reingold [13].

Lemma 20. Let G be a directed graph with n vertices and let f be a function of the entries in the adjacency matrix of G which has the value 1 if G contains a cycle, 0, otherwise. Then f is a nontrivial function of at least $n(n+1)/2$ entries in the adjacency matrix of G.

Proof: See [13]. QED

Lemma 21. Let (G,r) be a rooted graph, where $G = (V,E)$, with $n \geq 7$ vertices and let $i,j \in V$. Let $f_{(i,j)}$ be a function of the entries in the adjacency matrix of G which has the value 1, if i is a dominator of j in (G,r) and 0, otherwise. Then $f_{(i,j)}$ is a nontrivial function of at least $(n-1)(n-3)/4$ entries in the adjacency matrix of G.

Proof: The technique of proof is to find a rooted, directed graph G in which it is particularly hard, for some pair of vertices i,j, to determine if i is a dominator of j, from the adjacency matrix of G.

For $k \geq 5$, let $G_k = (V,E)$ be the directed graph with

$$V = \{r, u_1, \ldots, u_k, v_1, \ldots, v_k\}$$

and

$$E = \{\overrightarrow{(r,u_1)}, \overrightarrow{(r,v_1)}\} \cup \{\overrightarrow{(u_i,u_{i+1})}, \overrightarrow{(v_i,v_{i+1})} \mid i = 1, \ldots, k-1\}.$$

Then (G,r) is a rooted graph with $n = 2k + 1$ vertices. Let $i = v_1$ and $j = v_k$. Then i is a dominator of j in (G_k,r) thus $f_{(i,j)} = 1$. Let $U' = \{u_1, \ldots, u_k\}$ and $V' = \{v_2, \ldots, v_k\}$. Let A be the adjacency matrix of G_k. <u>Claim</u>: $f_{(i,j)}$ is a nontrivial function of $A(x,y)$ for each pair $(x,y) \in U' \times V'$. To see this, note that for $(x,y) \in U' \times V'$, $A(x,y) = 0$. Let A' be the $n \times n$ matrix where for $(u,v) \in V \times V$,

$$A'(u,v) = \begin{cases} 1, & \text{if } (u,v) = (x,y) \\ A(u,v), & \text{otherwise.} \end{cases}$$

Since i is a dominator of j in (G_k,r) but not in the graph with adjacency matrix A', $f_{(i,j)}(A) \neq f_{(i,j)}(A')$, proving the claim. Since $|U'| = (n-1)/2$ and $|V'| = (n-3)/2$, $f_{(i,j)}$ is a nontrivial function of at least $(n-1)(n-3)/4$ entries in the adjacency matrix.

<u>Lemma 22</u>. Let G be a connected, undirected, weighted graph with $n \geq 3$ vertices. Let (x,y) be an edge of G and let $f_{(x,y)}$ be a function of the weight matrix of G which has the value 1, if (x,y) is in a minimum spanning tree of G, and 0, otherwise. Then $f_{(x,y)}$ is a nontrivial function of at least $(n^2-5)/4$ entries in the weight matrix of G.

Proof: *The technique of this proof is similar to that of the* previous lemma. For $n \geq 3$, let $K_n = (V,E)$ be the complete graph on n vertices and let $(x,y) \in E$. Let $r = \lfloor n/2 \rfloor$ and $t = \lceil n/2 \rceil$. Partition V into two disjoint sets, $V_x = \{u_1, . . ., u_r\}$ and $V_y = \{v_1, ..., v_t\}$, of r and t elements, respectively, such that $x \in V_x$ and $y \in V_y$. For $z = x,y$, let S_z be the set of edges of a spanning tree of the complete subgraph of K_n on the vertices of V_z. Assign weights to the edges of K_n as follows. For $e \in E$,

$$w(e) = \begin{cases} 1, & \text{if } e \in S_x \cup S_y \\ 2, & \text{if } e = (x,y) \\ 3, & \text{otherwise.} \end{cases}$$

Then $S = S_1 \cup S_2 \cup \{(x,y)\}$ is the set of edges of a MST for K_n with weight function w and, in fact, is unique. Note that $f_{(x,y)} = 1$ since (x,y) is in a MST of K_n.

Let W be the weight matrix of K_n with weight function w.

Claim: *For each* $(u,v) \in S_x \times S_y$, such that $(u,v) \neq (x,y)$, $f_{(x,y)}$ is a nontrivial function of $W(u,v)$. To see this, let $(u',v') \in S_x \times S_y$, $(u',v') \neq (x,y)$, and let W' be the n x n matrix defined by

$$W'(u',v') = \begin{cases} 1, & \text{if } (u',v') = (u,v) \\ \\ W(u',v'), & \text{otherwise.} \end{cases}$$

If G' is the graph whose weight matrix is W', (x,y) is not an edge of a MST of G'. Thus, $f_{(x,y)}(W) \neq f_{(x,y)}(W')$, proving the claim. Since

$|S_x| = \lfloor n/2 \rfloor$ and $|S_y| = \lceil n/2 \rceil$, $|(S_x x S_y) - \{(x,y)\}| = \lfloor n/2 \rfloor \lceil n/2 \rceil - 1 \geq (n^2-5)/4$. Thus, $f_{(x,y)}$ is a nontrivial function of at least $(n^2-5)/4$ entries in the weight matrix. QED

Lemmas 20 through 22 are combined with Theorem 12 of Section 8.1 to give the following lower bound results.

Theorem 15. On an unbounded parallel model, at least 2logn - c time units, for some constant c, are required to determine:

(i) for a given directed graph G with n vertices, whether G contains a cycle, using the adjacency matrix of G as input,

(ii) for a given rooted graph (G,r), where G = (V,E), with $n \geq 5$ vertices and for $i,j \in V$, whether i is a dominator of j in (G,r) using the adjacency matrix of G as input, and

(iii) for a given connected, undirected, weighted graph G, with $n \geq 3$ vertices, the weight matrix of a MST of G, using the weight matrix of G as input.

Proof: (i) Any algorithm, which determines whether or not G contains a cycle, computes a function f, where f = 1 if G contains a cycle, and f = 0, otherwise. By Lemma 20, f is a nontrivial function of n(n+1)/2 input variables. Thus by Theorem 12, at least $\lceil \log(n(n+1)/2) \rceil$ time units are required to compute f from the adjacency matrix of G. The theorem follows by noting that

$$\lceil \log(n(n+2)/2) \rceil \geq \log n + \log(n+1) - 2 \geq 2 \log n - 2.$$

(ii) This follows as in (i) with the observation that

$$\lceil \log(n-1)(n-3)/4) \rceil \geq \log(n-1) + \log(n-3) - 2 \geq 2 \log n - 4,$$

for $n \geq 5$.

(iii) This follows as in (i) by observing that

$$\lceil \log(n^2-5)/4 \rceil \geq \log(n^2-5) - 2 \geq 2 \log n - 4.$$

for $n \geq 3$.

Chapter 9

CONCLUSION

It has been demonstrated in this paper that in solving certain graph problems, operations can be performed in parallel to substantially reduce the computation time. Using the unbounded parallel model of computation of Section 1.1, parallel algorithms of time complexity $O(\log^2 n)$ have been developed to solve the problems of finding minimum spanning trees, spanning trees, biconnected components, dominators, bridges, bridge connected components, cycles, cycle bases, and shortest cycles. In contrast, the best sequential algorithms for all of these problems, except finding a cycle basis, have a worst case time complexity of $O(n^2)$. The best sequential algorithm for finding a cycle basis has time complexity $O(n^3)$.

When the number of processors, $P(n)$, used by a parallel algorithm to achieve a time complexity, $T(n)$, is considered, it is possible that much waste is involved, that is, that $P(n) \cdot T(n)$ is much greater than the time complexity of the best serial algorithm. It has been shown that the algorithms presented here for finding minimum spanning trees, bridges, and cycle bases are not only "fast," but involve relatively little waste as well.

In Chapter 8, lower bounds of $2\lceil \log n \rceil + c$ were obtained for some of the graph problems listed above. Thus, several algorithms presented in this paper have time complexities which are optimal, at least to within a factor of $\log n$.

The results in this paper indicate that graph theoretic problems are capable of being solved efficiently on parallel computers. However, the design of a specific computer will severely influence the time required to solve a given problem. An analysis of the techniques used in the algorithms which have been presented here may produce ideas as to what design characteristics may be desirable in a parallel computer which must solve a large number of graph problems.

REFERENCES

1. Aho, Hopcroft, and Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley, Reading, Mass., 1974.

2. Arjomandi, Eshrat, "A Study of Parallelism in Graph Theory," Ph.D. Thesis, Department of Computer Science, University of Toronto, TR 86, Dec. 1975.

3. Barnes, G. H., Brown, R. M., Kato, M., Kuck, D. J., Slotnick, D. K., and Stoker, R. A., "The Illiac IV Computer," IEEE Trans. on Computers, vol. C-17, 1968, pp. 746-757.

4. Biggs, Norman, Algebraic Graph Theory, Cambridge University Press, New York, 1974.

5. Borodin, A. and Munro, I., The Computational Complexity of Algebraic and Numeric Problems, American Elsevier Publishing Company, Inc., New York, 1975.

6. Chandra, Ashok K., "Maximal Parallelism in Matrix Multiplication," IBM Rep. RC 6193, Sept. 1976.

7. Cray Research, Inc., "Cray-1 Computer," Chippewa Falls, Wis., 1975.

8. Csanky, L., "Fast Parallel Matrix Inversion Algorithms," SIAM J. Computing, vol. 5, no. 4, Dec. 1976, pp. 618-623.

9. Harary, Frank, Graph Theory, Addison-Wesley, Reading, Mass., 1969.

10. Hintz, R. G. and Tate, D. P., "Control Data STAR-100 Processor Design," COMPCON-72 Digest of Papers, IEEE Comp. Soc., 1972, pp. 1-4.

11. Hirschberg, D. S., "Parallel Algorithms for the Transitive Closure and the Connected Component Problems," Proc. 8th Annual ACM Symposium on Theory of Computing, Hershey, Pa., May 1976.

12. Hirschberg, D. S., "Fast Parallel Sorting Algorithms," submitted for publication, 1977.

13. Holt, Richard C. and Reingold, Edward M., "On the Time Required to Detect Cycles and Connectivity in Graphs," Math. Systems Theory, vol. 6, no. 2, 1972, pp. 103-106.

14. Hopcroft, John and Tarjan, Robert, "Efficient Planarity Testing," J. ACM, vol. 21, no. 4, Cot. 1974, pp. 549-568.

15. Marimont, R. B., "A New Method of Checking the Consistency of Precedence Matrices," J. ACM, vol. 6, 1959, pp. 164-171.

16. Muller, David E. and Preparata, Franco P., "Restructuring of Arithmetic Expressions for Parallel Evaluation," J. ACM, vol. 23, no. 3, July 1976, pp. 534-543.

17. Munro, Ian and Paterson, Michael, "Optimal Algorithms for Parallel Polynomial Evaluation," J. Comput. Sys. Sci., vol. 7, 1973, pp. 189-198.

18. Preparata, F. P., "Parallelism in Sorting," submitted to International Conference on Parallel Processing, August 1977.

19. Preparata, F. P. and Sarwate, D. V., "An Improved Parallel Processor Bound in Fast Matrix Inversion," submitted to Info. Proc. Let., 1977.

20. Prim, R. C., "Shortest Connection Networks and Some Generalizations," Bell Syst. Tech. J., vol. 36, 1957, pp. 1389-1401.

21. Reingold, Edward M., Nievergelt, Jurg, and Deo, Narsingh, Combinatorial Algorithms: Theory and Practice, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1977.

22. Rivest, Ronald L. and Vuillemin, Jean, "A Generalization and Proof of the Aanderaa-Rosenberg Conjecture," Proc. 7th Annual ACM Symposium on Theory of Computing, Albuquerque, N.M., May 1975.

23. Rosenberg, A. L., "On the Time Required to Recognize Properties of Graphs: A Problem," SIGACT News, vol. 5, 1973.

24. Rudolph, J. A., "A Production Implementation of an Associative Array Processor - Staran," AFIPS Fall 1972, AFIPS Press, Montvale, N.J., vol. 41, pt. 1, pp. 229-241.

25. Ruggiero, J. F. and Coryell, D. A., "An Auxiliary Processing System for Array Calculations," IBM Sys. J., vol. 8, 1969, pp. 118-135.

26. Savage, Carla, "A Parallel Algorithm for Finding Minimum Spanning Trees," submitted for publication, 1977.

27. Strassen, V., "Gaussian Elimination is Not Optimal," Num. Math., vol. 13, pp. 354-356.

28. Tarjan, R., "Depth-first Search and Linear Graph Algorithms," SIAM J. Computing, vol. 1, no. 2, 1972, pp. 146-160.

29. Tarjan, Robert, "Finding Dominators in Directed Graphs," SIAM J. Computing, vol. 3, no. 1, 1974, pp. 62-89.

30. Tarjan, R. Endre, "A Note on Finding the Bridges of a Graph," Info. Proc. Let., vol. 2, 1974, pp. 160-161.

31. Wulf, W. A. and Bell, C. G., "C.mmp, a Multi-mini-processor," AFIPS Fall 1972, AFIPS Press, Montvale, N.J., vol. 41, pt. 2, pp. 765-777.

32. Yao, Andrew Chi-chih, "An $O(|E| \log\log |V|)$ Algorithm for Finding Minimum Spanning Trees," Info. Proc. Let., vol. 4, no. 1, pp. 21-23.

VITA

Carla Savage was born on Novmeber 11, 1951, in Baltimore, Maryland. She received a B.S. in Mathematics from Case Western Reserve University in 1973 and an M.S. in Mathematics from the University of Illinois in 1975. From 1973 to 1976, she was a teaching assistant and, from 1976 to 1977, a research assistant at the University of Illinois.