Fast and Efficient Solution of Path Algebra Problems

VICTOR PAN*

Computer Science Department, State University of New York at Albany, Albany, New York 12222

AND

JOHN REIF†

Mathematical Science Research Institute, Berkeley, California 94720; and Computer Science Department, Duke University, Durham, North Carolina 27706

This paper extends the author's parallel nested dissection algorithm (Pan and Reif, Technical Report 88-18, Computer Science Department, SUNY Albany, 1988) originally devised for solving sparse linear systems. We present a class of new applications of the nested dissection method, this time to path algebra computations (in both cases of the single source path problems and of the all pair path problems), where the path algebra problem is defined by a symmetric matrix A whose associated undirected graph G has a known family of separators of small size $s(n)$ (in many cases of interest, $s(n) = O(\sqrt{n})$.) The assumption that G has known separators is reasonable in a large variety of practical dynamic situations, where G is fixed and the entries of the matrix A associated with the edges of G may vary with the input. We substantially improve the known algorithms for path algebra problems of this general class:

Previous estimates

Path problem	Sequential time	Parallel time	Processors	Precomputed separators
Single source	$O(n^2)$	$O(n)$	n	No
		$O(I(n)\log n)$	$n^3/(I(n)\log n)$	Yes
All pairs	$O(n^3)$	$O(n)$	n^2	No
		$O(I(n)\log n)$	$n^3/(I(n)\log n)$	Yes

* Supported by NSF Grants MCS 8203232 and DCR-8507573.

† This work was supported by Office of Naval Research Contract N00014-80-C-0647 and by NSF Grant DCR-8503151.

Preprint of paper appearing in Journal of Computer and Systems Sciences, Vol. 38, No. 3, June 1989, pp. 494-510.

New Algorithms

Path problem	Sequential time	Parallel time	Processors	Precomputed separators
Single source	$O(n^{1.5})$	$O((\log n)\sqrt{n})$	$n/\log n$	No
		$O(I(n)\log^2 n)$	$n^{1.5}/(I(n)\log n)$	Yes
All pairs	$O(n^2 \log n)$	$O((\log n)\sqrt{n})$	$n^{1.5}$	No
		$O(I(n)\log^2 n)$	$n^2/(I(n)\log n)$	Yes

Here we assume that G is given with its $O(\sqrt{n})$-separator family. The latter assumption can be lifted for the sequential time estimates and for $O((\log n)\sqrt{n})$ parallel time estimates for planar graphs, because the evaluation of an $O(\sqrt{n})$-separator family can be done in $O(n)$ sequential time (Lipton and Tarjan, *SIAM J. Appl. Math.* **36**, No. 2, (1979), 177–189) or, on PRAM, in $O((\log n)\sqrt{n})$ parallel time with $(\sqrt{n})/\log n$ processors (Gazit and Miller, manuscript, Computer Sci. Dept., University of Southern California, 1986), with small overhead constants in both cases. $I(n)$ denotes parallel time of computing the sum of n values, $I(n) = O(\log n)$ for any EREW PRAM, $I(n) = O(1)$ on a randomized CRCW PRAM. Furthermore using the randomized algorithm of (Gazit and Miller, *in* "Proceedings, 28th Annu. IEEE Symp. FOCS, 1987," pp. 238–248), we may precompute separators of a planar graph using $O(\log^2 n)$ time, $n + f^{1+\varepsilon}$ processors for a positive ε where f is the number of faces; this is *less* than the cost of the subsequent path computation. Moreover, we preserve the above processor bounds but further decrease the parallel time by a factor of $\log n$ (via a modification of our new algorithms based on pipelining) in the important case of computing the minimum cost paths in a planar graph. Further applications lead, in particular, to computing a maxflow and a mincut in an undirected planar network using $O(I(n)\log^2 n)$ parallel steps, $n^{1.5}/I(n)\log n$ processors, versus the known bounds, $O(\log^2 n)$ and n^4, of (Johnson, *J. ACM* **34**, No. 4 (1987), 950–967).

1. Introduction

In this paper we substantially improve the known parallel algorithms for several problems of practical interest that can be reduced to path algebra computations. The applications of path algebras include the problems of: vehicle routing, investment and stock control, dynamic programming with discrete states and discrete time, network optimization, artificial intelligence and pattern recognition, labyrinths and mathematical games, encoding and decoding of information, see Section 2 and also containing some further bibliography. References [24, 25, 6] present several general sequential algorithms for such problems based on matrix operations in *dioids* (*semirings*); see our next sections. These algorithms, however, do not seem efficient in the case of sparse input graphs. In the special case of the shortest path problems there exist even more effective sequential algorithms [4], but they have been extended neither to the case of general path algebra problems over dioids nor to the case of parallel computations. We propose a substantial improvement of these general algorithms in the important case where the input

matrix A is associated with a fixed undirected planar graph or, more generally, with an undirected graph from the class of graphs having known small separator families, see Definition 2 in Section 3.

Our improvement relies on our extension of the generalized nested dissection parallel algorithm of [16] to path algebra problems and on a further acceleration of that algorithm. Originally that parallel algorithm was applied to linear systems of equations, as an extension of the sequential algorithm of [12] for the same problem; then, in [17], it was extended to the linear least-squares problem and to the linear programming problem. In [12] the authors suggested (but apparently nowhere developed in any detail) the idea of the extension of their generalized nested dissection algorithm to path algebra computations for sparse graphs.

The extension of the generalized nested dissection algorithm to the case of dioids was somewhat surprising, because the divisions and subtractions of the original algorithm of [12] (and also of [16]) were not generally allowed in dioids.

Indeed, the algorithms of [12, 16] rely on factorization of the input matrix A (on the Cholesky factorization in [12] and on a special recursive factorization in [16]). We cannot extend such factorization to the case of dioids, due to the lack of subtractions and divisions, but we had extended the special recursive factorization of the inverse matrix A^{-1} of [16] to the similar factorization of the *quasi-inverse* A^*; and this turned out to suffice in many path algebra computations. (The definition of the *quasi-inverse* A^* generalizes the definition of the inverse matrix $(I-A)^{-1}$ to the case of dioids.) A^* can be computed via repeated squaring of A or of $I \oplus A$ (here and hereafter $\oplus$ denotes an addition in the dioid, I denotes the identity matrix); but using the recursive factorization of A^*, we improve even those simple computations in the case of sparse and well-structured input matrices A. Generally A^* is a dense matrix, even if A is sparse, so (unlike [6] and like [12, 16, 17]) we avoid explicitly computing A^* and exploit its (recursive) factorization. This makes the computations particularly effective where we solve the single source path problems.

We will define the algorithms over dioids (semirings); respectively, we will estimate the computational cost in terms of dioid operations. We assume a customary machine model of parallel computation, where in every parallel step each processor performs at most one operation of the considered class, see [2]; in our case this means at most one operation of the dioid. In the major specific applications to the classes of the problems of path existence, optimization, and counting, an operation over a dioid is an addition, a multiplication, or a comparison of two numbers; the numbers involved in the computation by our algorithms are usually represented with about the same precision (that is, with about the same number of binary digits) as the input values.

The table in our abstract shows our substantial improvement of the known algorithms in both sequential and parallel settings. The estimates of that table hold in the case of general path algebra problems for undirected planar graphs (as well as for all general undirected graphs) given with their $s(n)$—separator families where $s(n) = O(\sqrt{n})$ (the concept of $s(n)$-separator families will be formally defined in

Section 3). These estimates can be extended to the case where $s(n)$ is an arbitrary function, say $s(n) = O(n^{\sigma})$, $\sigma < 1$.

The chief assumption that we need in order to support our polygarithmic parallel time-complexity estimates is that the separator family is assumed given or readily computable. *For a planar graph its $O(\sqrt{n})$-separator family can be computed in $O(n)$ sequential time* [13], *or, on PRAM, in $O((\log n)\sqrt{n})$ parallel time using $(\sqrt{n})/\log n$ processors* [5], with small overhead constants in both algorithms [13, 5]; *or in $O(\log^2 n)$ time with $n + f^{1+\varepsilon}$ processors for any positive ε*, where f is the number of faces of the graph [5a]. The latter algorithm requires randomization at its auxiliary stage of computing a maximal independent set; for a grid graph a desired separator family is immediately available. Similarly we may assume the separator family preprocessed, say, for several computations for a graph with fixed sets of edges of variable lengths. The latter assumption is very reasonable in the case of many large-scale operations research problems, where the underlying graph G is fixed, but the costs associated with its edges (that is, the entries of the matrix A) may vary dynamically with the input. In particular, such a dynamic situation arises in an important case of a large computer network, where the costs of the links may vary in time and, moreover, may grow to infinity but where no new links are dynamically created. Another example is the commodity transportation problems, where the transportation links are known and fixed but may have dynamically changing costs. Furthermore these costs may grow so exorbitant that for some of those links they may be essentially infinite. In both of these examples, it is certainly useful to preprocess the underlying structure of the network in order to increase the efficiency of the resulting dynamic algorithm. In particular, in such cases we shall presume a precomputation stage where we shall find separators for the underlying graph and for a certain family of its subgraphs. This will lead us to a significant improvement of the dynamic path algebra computations of concern here.

All our parallel algorithms have polylogarithmic parallel time, that is, $O(I(n)\log^3 n)$ or $O(I(n)\log^2 n)$, and have processor bounds less than the known sequential time bounds for the same problems (which places our algorithms in *NC*, compare [2]). Here and hereafter $I(n)$ denotes the parallel time required for computing the sum of n values; $I(n) = O(\log n)$ for any EREW PRAM and $I(n) = O(1)$ on a randomized CRCW PRAM, see [21].

The bounds of the table in our summary can be applied to the general path computation in an undirected planar graph (network) G. This does not improve the known complexity bounds for some specific problems, such as both parallel and sequential complexity bounds for computing the transitive closure of a graph and the sequential time bounds of $O(n\sqrt{\log n})$ (single source) and $O(n^2)$ (all pairs) [4], for the shortest path problems. However, in many other cases we substantially improve the known estimates. In particular, we use only $O(I(n)\log^2 n)$ parallel steps and $n^{1.5}/(I(n)\log n)$ processors in order to solve the single source shortest path problem or $n^2/(I(n)\log n)$ processors in order to solve the all pair shortest path problem, whereas the known polylogarithmic time parallel algorithms for both

all pair shortest path problem and single source shortest path problem in planar graphs required n^3 processors, the same number as for the path computation in general graphs, compare the table in our summary. Furthermore, our parallel algorithms for the shortest path computations in planar graphs, combined with the results of [9, 10], lead to a new parallel algorithm for the evaluation of a maximum flow and a minimum cut in G using $O(I(n)\log^3 n)$ steps and $n^{1.5}/(I(n)\log n)$ processors or, alternatively, $O(I(n)\log^2 n)$ steps, $(n/\log n)^2$ processors, to compare with the previous bounds of $O(\log^2 n)$ steps, n^4 processors, [9]. To yield such an extension to computing maxflow-mincut in a planar undirected network, we need to use a simple extension of our path algebra results to the case of directed graphs. (Such an extension may have other applications to maxflow-mincut computation and may be itself of independent interest.) Several further applications can be expected; for instance, we may apply our parallel shortest path algorithms to feasibility testing of a multicommodity flow in a planar network in the case where all the k source-sink pairs lie on the boundary of the outer face; compare [4, 7, 14]. In that case the feasibility test based on our parallel shortest path computation and on the algorithm of [10] for constructing the auxiliary dual graphs, both applied within the construction of [14], requires only $O(I(n)\log n)$ parallel steps and $\min\{kn^{1.5}/(I(n)\log n), n^2/(I(n)\log n)\}$ processors, where k is the number of commodities. This improves the previously known processor bounds by more than on the factor of n.

Furthermore, in [19], for a special but important path algebra problem of computing the minimum cost paths in a (planar) graph, we use pipelining in a non-trivial way in order to rearrange our original parallel algorithms of [16–18] and to accelerate the computation by a factor of $\log s(n)$ (which means a factor of $\sigma \log n + C$ for $s(n) = Cn^{\sigma}$, $\sigma = 0.5$ for planar graphs); simultaneously, we preserve the original processor bounds (defined up to within a constant factor). This result also leads to the respective acceleration of parallel computation for mincut and maxflow and for other related problems.

In the next section we will introduce some preliminary technical definitions required for our parallel algorithms; in particular, we will define dioids and will state some path algebra problems; we will also estimate the computational cost of solving those problems in the case of general graphs. In Section 3 we will consider those problems for input graphs having small separator families and will present our main results, that is, our improvements of the known algorithms. Our parallel algorithms for the shortest path computations are applied to computing maxflow and mincut in Section 4.

2. Path Algebra Problems for General Graphs

In this section we will recall some auxiliary results and definitions for path algebra problems for general graphs. These definitions and results will be essential

to our later work, but those readers who are already familiar with these technical definitions and results should proceed to Section 3.

We will start with the special case of the shortest path problem in a graph $G = G(A)$ with n vertices defined by an $n \times n$ matrix $A = [a_{ij}]$ of nonnegative arc lengths where $a_{ij} = \infty$ if there is no arc between the vertices i and j in G. (A is symmetric if G is undirected.) We seek the vector $\bar{\mathbf{x}} = [x(i)]$ of distances $x(i)$ (that is, of the lengths of shortest paths) from vertex 1 to all vertices i in G. This is the *single source shortest path problem* (SS). The distances satisfy the following system of equations, $x(1) = 0$, $x(i) = \min_j (x(j) + a_{ji})$, $i = 2, ..., n$, or equivalently,

$$x(1) = \min(\min_j (x(j) + a_{j1}), 0), \qquad x(i) = \min(\min_j (x(j) + a_{ji}), \infty), \qquad i = 2, ..., n.$$

We substitute $\oplus$ for min (a noninvertible operation!) and $*$ for $+$ and rewrite this system as follows,

$$x(1) = \bigoplus_j (x(j) * a_{j1} \oplus 0), \qquad x(i) = \bigoplus_j (x(j) * a_{ji} \oplus \infty), \qquad i = 2, ..., n,$$

or, in matrix notation, denoting $\bar{\mathbf{i}}^{(1)} = [0, \infty, ..., \infty]$,

$$\bar{\mathbf{x}} = \bar{\mathbf{x}} * A \oplus \bar{\mathbf{i}}^{(1)}. \tag{1}$$

Here and hereafter we always assume that the operation $*$ preceeds $\oplus$, unless a different order is set up by parentheses.

Similarly, seeking the matrix $X = [x(i, j)]$ of distances between all pairs of vertices in G (this is *the all pair shortest path problem*, AP) and denoting $I = [\delta_{ij}]$, $\delta_{ii} = 0$, $\delta_{ij} = \infty$ if $i \neq j$, we arrive at the following matrix equation,

$$X = X * A \oplus I. \tag{2}$$

Restricting (2) to the hth row we arrive at the SS of computing the distances from the vertex h to all the vertices in G (for $h = 1$ we again arrive at (1)), so an AP can be reduced to n SSs.

Some known algorithms of linear algebra can be extended to solve systems (1) and (2); this may turn them into known combinatorial algorithms for the SS and/or the AP. Here are two examples [24, 25, 6].

ALGORITHM 1. Set $\bar{\mathbf{x}}^{(0)} = \bar{\mathbf{i}}^{(1)}$; compute $\bar{\mathbf{x}}^{(k+1)} = \mathbf{x}^{(k)} * A \oplus \bar{\mathbf{i}}^{(1)}$, $k = 0, 1, ...$ until $\bar{\mathbf{x}}^{(k+1)} = \bar{\mathbf{x}}^{(k)}$; then output the vector $\bar{\mathbf{x}} = \bar{\mathbf{x}}^{(k)}$ satisfying (1).

Algorithm 1 extends Jacobi's method of linear algebra and amounts to the algorithm of [1] for the SS.

ALGORITHM 2. (a) Set $A^{[0]} = A$.

(b) For $k = 0, 1, ..., n-1$, compute $a_{ij}^{[k+1]} = a_{ij}^{[k]} \oplus a_{ik}^{[k]} * a_{kj}^{[k]}$, $i, j = 1, ..., n$.

(c) Output $X = A^{[n]} \oplus I$. (The matrix X satisfies (2).)

Algorithm 2 extends Jordan's algorithm of linear algebra and amounts to the algorithm of [3] for the AP, compare also Algorithm 4 below (in Section 3).

Several other path computation problems can be also reduced to the solution of the linear systems (1) or (2) or to some similar matrix computations performed by means of additions and multiplication only. We need to recall a general concept, already implicitly used in our reduction of the SS to (1) and of the AP to (2).

DEFINITION 1. A *dioid* (sometimes called a *semiring*, because it extends noncommutative rings to the case where subtractions may not be defined) is a set S with two operations, $\oplus$ and $*$, such that for any triple of elements $a, b, c \in S$ and for two special elements e (unity) and ε (zero) of S, the following equations hold:

$$a \oplus b = b \oplus a \in S, \qquad (a \oplus b) \oplus c = a \oplus (b \oplus c), \qquad a \oplus \varepsilon = a,$$

$$a * b \in S, \qquad (a * b) * c = a * (b * c),$$

$$a * e = e * a = a, \qquad a * \varepsilon = \varepsilon * a = \varepsilon,$$

$$a * (b \oplus c) = (a * b) \oplus (a * c), \qquad (b \oplus c) * a = (b * a) \oplus (c * a).$$

In the above reduction of the SS to (1) and of the AP to (2), we used the dioid where $S = \mathbf{R} \cup \{\infty\}$, $\mathbf{R}$ being the set of real numbers, $\oplus = \min$, $* = +$, $e = 0$, $\varepsilon = \infty$. (This dioid is also used for other optimization path problems, see (iii) below.) Generalizing (1) and (2) to arbitrary dioids, we define that

$$\bar{\mathbf{1}}^{(1)} = [e, \varepsilon, ..., \varepsilon], \qquad \mathbf{I} = [\delta_{ij}], \qquad \delta_{ii} = e, \ \delta_{ij} = \varepsilon \text{ if } i \neq j. \tag{3}$$

Here is a list of some classes of path problems, which can be reduced to solving the systems (1) and (2) or to similar matrix operations in appropriate dioids:

(i) existence (problems of graph connectivity);

(ii) enumeration (elementary paths, multicriteria problems, generation of regular languages);

(iii) optimization (paths of maximum capacity, paths with minimum number of arcs, shortest paths, longest paths, paths of maximum reliability, reliability of a network);

(iv) counting (counting of paths, Markov chains).

Specifically, the class (i) includes the problems of

(a) the existence of paths having k (or at most k) arcs between vertices i and j in a given (*di*)graph G (for a fixed k);

(b) computing the transitive closure of G;

(c) testing G for being strongly connected and for having circuits.

An appropriate dioid for problems of class (i) is the Boolean algebra, $S = \{0, 1\}$, $\oplus = \max$, $* = \min$, $\varepsilon = 0$, $e = 1$; in the incidence matrix $A = [a_{ij}]$ of G, $a_{ij} = 1$ if and only if $\{i, j\}$ is an arc of G.

The subclass of shortest path problems in (iii) includes SS, AP (also in the versions where the shortest paths are required to have exactly k or at most k arcs), and testing a graph for having circuits of negative lengths.

Class (iv) includes counting the numbers of

(a) distinct paths having k (or at most k) arcs between i and j in G;

(b) all the distinct paths between i and j in G.

In the dioid for this class, S is the set of integers, $\oplus = +$, $* = *$ (that is, $\oplus$ and $*$ are the conventional addition and multiplication, respectively), $\varepsilon = 0$, $e = 1$; $A = [a_{ij}]$, $a_{ij} = 1$ if and only if $\{i, j\}$ is an arc of G.

The solution of most of the path problems listed in the previous section can be reduced to the evaluation (over the dioid) of the entries of the matrix $A^{(k)}$ (the all pair path problems) or of the vector $\bar{\mathbf{b}}A^{(k)}$ (the single source path problems) for some positive k, usually for $k = n - 1$. Here

$$A^{(q+1)} = A^{(q)} \oplus A^{q+1}, \qquad q = 0, 1, ...,$$

$A^{(0)} = I$ (see (3)), A is an $n \times n$ input matrix, $\bar{\mathbf{b}} = \bar{\mathbf{i}}^{(h)}$ is a fixed coordinate vector of dimension n. Here and hereafter we assume that all computations, in particular, computing matrix sums, products, and powers, are performed over the dioid associated with a given path problem; we simplify the notation, writing $\bar{\mathbf{u}}U$ and UV (rather than $\bar{\mathbf{u}} * U$ and $U * V$, respectively) in order to denote the product of a vector $\bar{\mathbf{u}}$ by a matrix U and the product of matrices U and V over dioids and similarly for matrix powers over dioids.

There exists the *quasi-inverse* matrix, defined as

$$A^* = \lim_{q \to \infty} A^{(q)}, \tag{4}$$

for the incidence matrix A of each of the path problems listed in the previous section, except for those shortest path and multicriteria problems where there exist circuits of negative lengths in G and for those counting problems where there exists a circuit in G. In both latter cases the existence of such circuits is detected by computing A^k or $(I \otimes A)^k$ over the dioids. Hereafter we will consider only the most typical case, where there exists the quasi-inverse A^* and where, moreover,

$$A^* = A^{(n-1)}, \qquad A^{(q)} = A^{(q-1)} \qquad \text{for} \quad q \geqslant n. \tag{5}$$

(Our estimates of this section for the cost of the evaluation of A^* and $\bar{\mathbf{b}}A^*$ under (5) can be immediately extended to the case of the evaluation of A^q, $A^{(q)}$, and $\bar{\mathbf{b}}A^{(q)}$ for $q \neq n - 1$.) Equations (4) and (5) imply that A^* is the incidence matrix of the transitive closure of the graph of A for several problems of connectivity, existence, and optimization. Equation (5) implies that

$$A^* = I \oplus A \oplus A^2 \oplus \cdots \oplus A^{n-1},$$

so A^* can be computed as

$$A^* = (I \oplus A)(I \oplus A^2)(I \oplus A^4) \cdots (I \oplus A^{2^k}), \qquad k = \lceil \log_2 n \rceil, \tag{6}$$

using a total of $(4nk - k + 1)\, n^2$ operations in the dioid. (The known fast matrix multiplication algorithms, see [15], cannot be generally applied over dioids.) For many dioids (including the dioid that we associated with the shortest path computations) the operation $\oplus$ is *idempotent*, that is, $a \oplus a = a$ for all $a \in S$. In that case

$$A^* = \bigoplus_{r=0}^{n} A^r = \bigoplus_{r=0}^{n} C(n, r)\, A^r = (I \oplus A)^n = (I \oplus A)^{2^k} \tag{7}$$

(where $C(n, r) = r!/n!(n-r)!$, $k = \lceil \log_2 n \rceil$), so A^* can be computed via repeated squaring of $I \oplus A$, using only $n^2(k-1)(2n-1) + n$ dioid operations. The resulting asymptotic estimates for the cost of both algorithms (6) and (7) are the same: $O(n^3 \log n)$ operations or $O(I(n) \log n)$ parallel steps, $\lceil n^3/I(n) \rceil$ processors. The algorithms (6) and (7) are not quite efficient if A is sparse, for the sparsity of A is not generally preserved during the computation. We may, of course, compute $\bar{\mathbf{b}} A^*$ via computing A^*; alternatively, we may perform n successive postmultiplications of the vectors $\bar{\mathbf{b}} \sum_{r=0}^{k} A^r$ by the matrix A for $k = 0, 1, \ldots, n-1$ and $n-1$ vector additions.

3. Solving Path Problems for Graphs with Small Separators

3.1. *Algorithms*

Definition 2 [16]. A graph $G = (V, E)$ is said to have an *$s(n)$-separator family* (with respect to two constants, $\alpha < 1$ and a natural n_0) if either $|V| \leqslant n_0$ or deleting some separator set of vertices S of cardinality $|S| \leqslant s(|V|)$, we may partition G into two disconnected subgraphs with the vertex sets V_1 and V_2, such that $|V_i| \leqslant \alpha |V|$, $i = 1, 2$, and if, furthermore, each of the two subgraphs of G defined by the vertex sets $S \cup V_i$, $i = 1, 2$, also has an $s(n)$-separator family (with respect to the same constants α and n_0).

Grid graphs on a d-dimensional hypercube have $(n^{1-1/d})$-separator families, which are readily available; in particular, square grids have $\sqrt{n}$-separator families. An undirected planar graph has a $\sqrt{8n}$-separator family, which can be computed in $O(n)$ sequential time [13], or on PRAM in $O(\sqrt{n})$ parallel time using $\sqrt{n}$ processors [5] (with small overhead constants in both algorithms [13, 5]), or in $O(\log^2 n)$ randomized parallel time with $n + f^{1+\varepsilon}$ processors for any positive ε, where f is the number of faces [5a]. In many cases several computations must be performed for the same graph G, having, say variable edge weights; in such cases

one may precompute an $s(n)$-separator family of G, provided that there exists such a family.

In this section we will consider a path problem for an undirected graph $G=(V, E)$ given together with its $s(n)$-separator family, $s(n)=cn^{\sigma}$, $\frac{1}{3}<\sigma<1$, c is a positive constant. In this case we will further decrease the cost of the computation of A^* and $\bar{\mathbf{b}}A^*$ by extending the generalized nested dissection algorithms of [12] for sequential computation and of [16] for parallel computation.

The generalized nested dissection algorithms of [12] and [16] require inverting some auxiliary matrices of smaller sizes. Dioid elements and matrices in dioids may have no inverses, but in our extension of the algorithm of [16], we compute quasi-inverses of matrices over dioids applying either the algorithms (6), (7) of the previous section or the following generalized Jordan elimination algorithm, which requires only operations $\oplus$ and $*$ [24, 25, 6, p. 110]. (The latter algorithm uses $O(n)$ parallel time and n^2 processors or $O(n^3)$ operations in dioids.)

ALGORITHM 3 (evaluation of A^*). Set $A^{[0]}=A$, $B^{[0]}=I$ and recursively compute $A^{[k]}=M^{[k]}A^{[k-1]}$, $B^{[k]}=M^{[k]}B^{[k-1]}$ for $k=1, 2, ..., n$. Here $M^{[k]}$ is obtained from the matrix I of (3) by replacing the (k, k) entry of I by $(a_{kk}^{[k-1]})^*$ and by replacing other entries of the kth column of I by the entries of the vector $[a_{ik}^{[k-1]} * (a_{kk}^{[k-1]})^*]$, where $i, k=1, ..., n$. Output $B^{[n]}$.

Algorithm 3 extends (to dioids) the Jordan elimination scheme for computing $B^{-1}=(I-A)^{-1}$ via the solution of the linear system $BX=I$, which can be interpreted as n linear systems, each consisting of n equations in n unknowns, having the same coefficient matrix, $B=I-A$, and having one of the n unit coordinate vectors on the right side. We will use the following result.

LEMMA 1. *Let $B^{[n]}$ be the output matrix computed by Algorithm* 3 (*so that for all the auxiliary matrices $a_{kk}^{[k-1]}$ there exist the quasi-inverse matrices $(a_{kk}^{[k-1]})^*$; compare Remark* 1 *below*). *Then $B^{[n]}=A^*$.*

In the Appendix, Lemma 1 is proven using, in particular, the argument of [24, 25, 6, pp. 108–110] and the next simple lemma (which can be immediately verified, see (4)).

LEMMA 2. *The matrix $X=A^*$ satisfies* (2).

Next we will give a proof of Lemma 1 that is slightly longer than the one in the Appendix, but more direct and informative. In this proof we start with the customary Jordan elimination scheme over a field, say of real or rational numbers. In that scheme, n successive premultiplications by n matrices $M^{[k]}$ (for $k=1, ..., n$) of a special format (see below) reduce $n\times n$ input matrix $G=G^{[0]}$ to the identity matrix, so that $G^{[k]}=[g_{ij}^{[k]}]=M^{[k]}G^{[k-1]}$, $k=1, ..., n$, $M^{[n]}M^{[n-1]}\cdots M^{[1]}G=I$, $M^{[n]}M^{[n-1]}\cdots M^{[1]}=G^{-1}$. It can be immediately verified that the latter matrix

identities hold if the matrices $M^{[k]}$ are defined as follows. $M^{[k]} = [m_{ij}^{[k]}]$, where i, j, k range from 1 to n; $m_{ij}^{[k]} = 0$ if $i \neq j$, $j \neq k$; $m_{jj}^{[k]} = 1$ unless $j = k$; $m_{ik}^{[k]} = -g_{ik}^{[k-1]}/g_{kk}^{[k-1]}$ unless $i = k$; $m_{kk}^{[k]} = 1/g_{kk}^{[k-1]}$. These recursive expressions for $M^{[k]}$ through $G = G^{[0]}$ involve both subtractions and divisions. Now, in addition to the above sequences of matrices $\{G^{[k]}, M^{[k]}, k = 0, 1, ..., n\}$, consider also the sequence $\{A^{[k]}, k = 0, 1, ..., n\}$, where $A^{[0]} = A = I - G$, $A^{[k]} = [a_{ij}^{[k]}] = M^{[k]}A^{[k-1]}$ for $k = 1, 2, ..., n$. We will exploit the following correlation between the entries $a_{ik}^{[h]}$ and $g_{ik}^{[h]}$ for $h < k$ (which do not generally hold for $h \geqslant k$): $a_{ik}^{[h]} = -g_{ik}^{[h]}$ unless $i = k$ and $a_{kk}^{[h]} = 1 - g_{kk}^{[h]}$ for $h = 0, 1, 2, ..., k-1$ for all k, so $m_{ik}^{[k]} = a_{ik}^{[k-1]}/(1 - a_{kk}^{[k-1]})$ unless $i = k$, $m_{kk}^{[k]} = 1/(1 - a_{kk}^{[k-1]})$. Replacing $1/(1-a)$ by the formal power series $a^* = 1 + a + a^2 + \cdots$ for $a = a_{kk}^{[k-1]}$, $k = 1, ..., n$ (such a series converges to $1/(1-a)$ if $|a| < 1$), we arrive at Algorithm 3 for computing the matrices $M^{[1]}, M^{[2]}, ..., M^{[n]}$ over dioids and at the desired matrix identity

$$B^{[n]} = M^{[n]}M^{[n-1]} \cdots M^{[1]} = I \oplus A \oplus A^2 \oplus \cdots = A^*.$$

That identity involves only additions and multiplications, so it holds over dioids. Q.E.D.

$A^{[n]} = B^{[n]}A$ by the definition of $A^{[n]}$ and $B^{[n]}$ in Algorithm 3, therefore $A^* = A^{[n]} \oplus I$; so we may dispense with the evaluation of $B^{[k]}$ and simplify Algorithm 3 as follows.

ALGORITHM 4 (evaluation of A^*).

(a) Set $A^{[0]} = A$.

(b) For k from 1 to n,

$$a_{kk}^{[k]} = (a_{kk}^{[k-1]})^*$$

$$a_{ij}^{[k]} = a_{ij}^{[k-1]} \oplus a_{ik}^{[k-1]} * a_{kk}^{[k]} * a_{kj}^{[k-1]} \text{ for all } i, j \text{ except for } i = j = k$$

(c) Output $A^* = A^{[n]} \oplus I$.

Algorithm 4 turns into Algorithm 2 of Section 2 for the dioids where $a_{kk}^{(k)} = e$, which is the case, in particular, for the dioids associated with the shortest path problems.

Algorithms 3 and 4 can be applied to symmetric and nonsymmetric matrices A, but hereafter we will focus on the symmetric case, where the matrix A is associated with an undirected graph, $G(A)$.

To compute A^* in parallel, we *recursively factor* the matrix $A_0^* = (PAP^T)^*$ (this extends the recursive factorization from [16] for A_0^{-1}, the inverse of $A_0 = PAP^T$). Here P denotes the permutation matrix obtained in the ordering stage of the generalized nested dissection algorithm of [16] applied to the input matrix A. Here and hereafter W^T denotes the transpose of a matrix W; O denotes the null matrices,

filled with the zeros ε; and I denotes the identity matrices of (3) of appropriate size. Here is our recursive factorization, where $h=0, 1, ..., d-1$, $d=O(\log n)$:

$$A_h=\begin{bmatrix} X_h & Y_h^{\mathrm{T}} \\ Y_h & Z_h \end{bmatrix}, \qquad A_{h+1}=Z_h\oplus Y_hX_h^*Y_h^{\mathrm{T}}, \tag{8}$$

$$A_h^*=\begin{bmatrix} I & X_h^*Y_h^{\mathrm{T}} \\ O & I \end{bmatrix}\begin{bmatrix} X_h^* & O \\ O & A_{h+1}^* \end{bmatrix}\begin{bmatrix} I & O \\ Y_hX_h^* & I \end{bmatrix}. \tag{9}$$

Let us verify that (9) indeed defines A_h^*. Expand the right side of (9) deleting h and replacing $h+1$ by 1 in the subscripts, to simplify the notation,

$$W=\begin{bmatrix} X^*\oplus X^*Y^{\mathrm{T}}A_1^*YX^* & X^*Y^{\mathrm{T}}A_1^* \\ A_1^*YX^* & A_1^* \end{bmatrix}. \tag{10}$$

LEMMA 3 (see the proof in the Appendix and compare also Remark 2 below). *Let* $A=\left[\begin{smallmatrix} X & Y^{\mathrm{T}} \\ Y & Z \end{smallmatrix}\right]$, *let there exist the quasi-inverses* X^* *and* $A_1^*=(Z\oplus YX^*Y^{\mathrm{T}})^*$, *and let* W *be defined by* (10). *Then* $WA\oplus I=W$.

Similarly to the proof of Lemma 1 (see Appendix), we deduce from Lemma 3 that $W=A^*$ under (10) (provided that there exist the quasi-inverses A_1^*, X^*). This immediately substantiates the validity of the recursive factorization (8), (9). (Note that computing a^* for $a\in S$ may require more than one operation in a dioid unless (5) holds; on the other hand, we may compute a^* as $(e-a)^{-1}$ in the dioids that have inverse operations to $\oplus$ and $*$).

Remark 1. The proofs of Lemmas 1 and 3 and consequently of the validity of Algorithms 3, 4 and of the recursive factorization (8), (9) do not require us to assume (5). It is sufficient to use the definition (4) of a quasi-inverse and to assume the existence of all the quasi-inverses included in Algorithms 3, 4 and in the recursive factorization (8), (9). Furthermore, we may modify Algorithm 3 (including also the row and column interchanges) in order to ensure the existence of the quasi-inverses of all pivot entries (we may do this unless the quasi-inverse A^* does not exist for a given input matrix A).

Remark 2. The matrix factorization (9) can be computed using Algorithm 3. Indeed rewrite (9) as

$$A_h^*=\begin{bmatrix} I & X_h^*Y_h^{\mathrm{T}}A_{h+1}^* \\ O & A_{h+1}^* \end{bmatrix}\begin{bmatrix} X_h^* & O \\ Y_hX_h^* & I \end{bmatrix}.$$

Let $n=2$ and let A_h replace A in Algorithm 3. Then Algorithm 3 computes the latter factorization applied to the 2×2 block matrix A_h. (This application of Algorithm 3 is valid because matrix algebras constitute a special class of dioids.)

Remark 3. In [6] the fact that the solution $X=B^{[n]}$ of (2) equals A^* (see Lemma 1) is stated under the additional assumption that the preorder relation in

the dioid ($a \leqslant b$ if and only if there exists c such that $a \oplus c = b$) is the order relation ($a \leqslant b$ and $a \geqslant b$ together imply that $a = b$). Under that assumption, [6] suggests (with the proof omitted) that $B^{[n]}$ is the minimum solution. This implies that $B^{[n]} = A^*$ for A^* is easily proven to be the minimum solution to (2); both our proofs of Lemma 1 imply that $B^{[n]} = A^*$ even where such a preorder relation in the dioid is not assumed.

Remark 4. The matrix equations (8) and (9) generalize the recursive factorization from [16] based on the following factorization of $A_0 = PAP^{\mathrm{T}}$ over a field (which itself, however, does not seem to be extendable to the case of dioids),

$$A_h = \begin{bmatrix} X_h & Y_h^{\mathrm{T}} \\ Y_h & Z_h \end{bmatrix}, \qquad Z_h = A_{h+1} + Y_h X_h^{-1} Y_h^{\mathrm{T}},$$

$$A_h = \begin{bmatrix} I & O \\ Y_h X_h^{-1} & I \end{bmatrix} \begin{bmatrix} X_h & O \\ O & A_{h+1} \end{bmatrix} \begin{bmatrix} I & X_h^{-1} Y_h^{\mathrm{T}} \\ O & I \end{bmatrix}.$$

3.2. *Computational Cost Estimates*

Next we will estimate the cost of computing $\bar{\mathbf{b}}A^*$ and A^*. We only need to compute the quasi-inverses X_h^*, the products $Y_h X_h^*$ (which also gives X_h^*, $Y_h^{\mathrm{T}} = (Y_h X_h^*)^{\mathrm{T}}$), and the matrices $A_{h+1}^* = (Z_h \oplus Y_h X_h^* Y_h^{\mathrm{T}})^*$ for $h = 0, 1, \ldots, d-1$, $d = O(\log n)$. At first we will proceed similarly to [16]. Then for each auxiliary matrix C that denotes an $s \times s$ diagonal block of the matrices X_h (here $s \leqslant s(\alpha^h n)$, $\alpha < 1$; $h = k, k-1, \ldots, 0$, $k = O(\log n)$), we need to compute the vector $C^* \bar{\mathbf{v}}$ for a fixed vector $\bar{\mathbf{v}}$. We may extend the assumed property that $A^{(q+1)} = A^{(q)}$ for $q \geqslant n-1$ to the equations $C^{(q+1)} = C^{(q)}$ for $q \geqslant s(\alpha^h n) - 1$. (Indeed, A and C are associated with the path problems of the same kind, having only different sizes, n and $s(\alpha^h n)$, respectively.) For faster parallel evaluation of C^* given C, we may apply the algorithms (6), (7) cited in the dense matrix case. Then such algorithms will use $O(k(s)\, I(s))$ parallel steps, $\lceil 2s^3/(I(s) \log s) \rceil$ processors where $k(s) = \lceil \log_2 s \rceil$, $s = s(\alpha^h n)$. Alternatively we may compute C^* applying Algorithm 4; this would involve $O(s^3)$ (sequential) dioid operations or $O(s)$ parallel steps, s^2 processors (yielding a slower but slightly more efficient version of parallel algorithms). (In the latter case the computations are arranged similarly to the algorithm of [12], except that computing the quasi-inverses X_h^* replaces the Cholesky factorization of X_h for all h.)

Other arithmetic operations used in the algorithm of [16] are additions and multiplications, which are replaced by the similar dioid operations. Thus we arrive at the favorable complexity bounds of $O(I(n) \log^2 n)$ parallel steps and $s^3(n)/(I(n) \log n)$ processors for computing the recursive factorization (8), (9), and of $O(I(n) \log n)$ parallel steps and $((|E|/\log n) + s^2(n))/I(n)$ processors for computing $\bar{\mathbf{b}}A^*$ for every vector $\bar{\mathbf{b}}$, provided that the recursive factorization is already available. Here $|E|$ denotes the number of edges of the graph associated with the matrix A, $|E| = \mathbf{O}(n)$ for planar graphs. Therefore $O(I(n) \log^2 n)$ steps

suffice for both single source path problems (where the vector $\bar{\mathbf{b}}$ is fixed, and only the row $\bar{\mathbf{b}}A^*$, but not the whole matrix A^*, must be computed; so that $(|E|+s^3(n))/(I(n)\log n)$ processors suffice) and the all pairs path problem (where we compute A^*, say by evaluating $\bar{\mathbf{b}}A^*$ for all the n coordinate vectors $\bar{\mathbf{b}}$; in this case we use $n(|E|/\log n+s^2(n))/I(n)$ processors). In a slower parallel algorithm we need $O(s(n))$ parallel steps and neither $s(n)^2+|E|/s(n)$ processors to solve the single source path problem or $(|E|+s(n)^2\log n)n$ processors to solve the all pair path problem. By multiplying the latter parallel time and processor bounds together, we arrive at the sequential time bounds of $O(|E|+s^3(n))$ in the case of the single source path problem and of $O((|E|+s^2(n)\log n)n)$ in the case of the all pair path problem.

If $s(n)=O(\sqrt{n})$ and $|E|=O(n)$, as in the case of planar graphs, we arrive at the estimates shown in the table in our abstract.

4. Improvement of Parallel Evaluation of a Minimum Cut and of a Maximum Flow in an Undirected Planar Network

The best sequential algorithms for computing a minimum cut and a maximum flow in an undirected planar network $N=(G, c)$ ($G=(V, E)$ denotes a graph, c denotes a set of the edge capacities) run in $O(n\log n)$ time and exploit the reduction to the shortest path computations, see [4, 8, 20]. Specifically, [20] presented $O(n\log^2 n)$ time algorithm for computing a mincut, [8] extended that algorithm to computing a maximum flow, and [4] improved the time bound to $O(n\log n)$. The previous best parallel polylog time algorithms [9] for those problems (by means of parallelization of the sequential scheme) require on the order of n^4 processors using polylogarithmic time. Combining our results for the SS in planar graphs with the results of [10], we arrive at the bounds of $O(I(n)\log^3 n)$ parallel steps and $n^{1.5}/(I(n)\log n)$ or alternatively, $O(\log^3 n)$ and $(n/\log n)^2$. More precisely, we should use the extensions of our results for the SS in *planar digraphs*; such extensions for both SS and AS immediately follow if we replace the matrices Y_h^T for all h by general matrices W_h of the same sizes but defined independently of Y_h.

Reference [9] computes a mincut and the value $v_{\max}$ of a maxflow using the following stages (see [8–10, 20] for further details):

(1) compute a planar embedding of N, for the estimated cost of $O(\log^2 n)$ steps, n^4 processors;

(2) find the planar dual network $D(N)$; step, processor bounds are $O(\log n)$, n^3;

(3) compute the μ-path, that is, the shortest path in the dual between the two faces F_s and F_t that adjoin the source s and the sink t of the primal network; step, processor bounds are $O(I(n)\log n)$, n^3;

(4) compute the consistent clockwise orderings for the faces on the μ-path; step, processor count is $O(\log n)$, n^2;

(5) compute the F-minimum cut-cycles in $D(N)$ for every dual vertex F on the μ-path (that is, compute the cut-cycles of the minimum length in $D(N)$ passing through the vertex F); the latter stage can be reduced to solving the AP in the dual network $D(N)$; the cost is $O(I(n)\log n)$ steps, n^3 processors;

(6) finally, compute the minimum value of the F-minimum cut-cycles over all the dual vertices F on the μ-path; this gives $v_{\max}$ and a mincut; the cost is $O(\log n)$ steps, n processors.

The recent algorithm of [10] performs the computation in the substage (1) using $O(\log^2 n)$ steps, n processors; the computation also includes substages (2) and (4) performed using $O(\log n)$ steps, n processors. Applying our parallel algorithm for the SS in stage (3) and our parallel algorithm for the AP in stage (5), we perform the computations in those stages in $O(I(n)\log^2 n)$ steps using $n^{1.5}/(I(n)\log n)$ processors in stage (3) and $n^2/(I(n)\log n)$ in stage (5). Summarizing we arrive at $O(I(n)\log^2 n)$ steps, $(n/\log n)^2$ processors computing $v_{\max}$ and a mincut. Alternatively, we may use $O(I(n)\log^3 n)$ steps and $O(n^{1.5}/(I(n)\log n))$ processors in stage (5); this would dominate the overall complexity. To arrive at these bounds, we apply the algorithm of [20], which performs stage (5) by successively solving SSs in the dual networks derived from $D(N)$. This is performed in at most $\lceil \log_2 n \rceil$ substages; in substage r up to 2^r SSs are solved in the derived networks, having the total number of edges at most $2|E| + 2^r$, $r = 0, 1, \ldots$. Here E is the edge set of the original planar network, $|E| = O(n)$, $2^r \leqslant 2^{1+\log n} = 2n$, so the total number of edges in the derived network is $O(n)$ in each substage. Therefore our algorithm for the SS enables us to perform the computations in each substage using $O(I(n)\log^2 n)$ steps, $n^{1.5}/(I(n)\log n)$ processors, so $O(I(n)\log^3 n)$ steps, $n^{1.5}/(I(n)\log n)$ processors suffice in all substages of stage (5) and consequently for the entire computation of $v_{\max}$ and a mincut.

When a mincut (passing through a vertex F on the μ-path) and the value $v_{\max}$ are known, we may immediately reduce computing a maxflow to a SS in the dual network, following [8]. (Specifically, this is the SS of computing the shortest distances between F and all other vertices in the dual network N.) Thus in that final stage we only need $O(I(n)\log^2 n)$ steps, $n^{1.5}/(I(n)\log n)$ processors.

APPENDIX: Proof of Lemmas 1 and 3

Proof of Lemma 1. It is immediately verified that $B^{[n]} = A^*$ is the unique solution A^* of (2) in the case of the special dioid where S is the set of real matrices, $\oplus = +$, $* = *$, $\varepsilon = 0$, $e = 1$,

$$|a_{kk}^{[k-1]}| < 1 \qquad \text{for all } k, \tag{11}$$

and, say,

$$n \max_{i,j} |a_{ij}| < 1. \tag{12}$$

The latter inequality immediately implies that $A^* = \sum_{h=0}^{\infty} A^h$ converges to $(I-A)^{-1}$

whereas $|a_{kk}^{[k-1]}| < 1$ implies that $(a_{kk}^{[k-1]})^* = \sum_{h=0}^{\infty} (a_{kk}^{[k-1]})^h$ converges to $(1 - a_{kk}^{[k-1]})^{-1}$.

Let us consider again an arbitrary dioid $(S, \oplus, *)$, where v^* is defined as the formal power series, $\bigoplus_{h=0}^{\infty} v^h$, $v^0 = e$ if $v \in S$, $v^0 = I$ if v is a matrix with the entries from S. Then the entries of $B^{[n]}$ and A^* are multivariate power series in the entries of A. The numerical values of the two power series representing the (i, j)-entries of $B^{[n]}$ and A^* for an arbitrary pair i, j must coincide with each other on any real matrix A such that (11) and (12) hold. It follows that such two power series coincide with each other also as formal power series. Q.E.D.

Proof of Lemma 3. Let $X = X_h$, $Y = Y_h$, $Z = Z_h$ and let (3), (8), (10) define I, $A = A_h$, $A_1 = A_{h+1}$, W. Then the matrix $WA \oplus I$ has the upper left block

$$\begin{aligned} & I \oplus X^*X \oplus X^*Y^{\mathrm{T}}A_1^*YX^*X \oplus X^*Y^{\mathrm{T}}A_1^*Y \\ &\quad = (I \oplus X^*Y^{\mathrm{T}}A_1^*Y)(X^*X \oplus I) \\ &\quad = X^* \oplus X^*Y^{\mathrm{T}}A_1^*YX^*, \qquad \text{since} \quad X^*X \oplus I = X^*; \end{aligned}$$

has the upper right block

$$\begin{aligned} & X^*Y^{\mathrm{T}} \oplus X^*Y^{\mathrm{T}}A_1^*YX^*Y^{\mathrm{T}} \oplus X^*Y^{\mathrm{T}}A_1^*Z \\ &\quad = X^*Y^{\mathrm{T}} \oplus X^*Y^{\mathrm{T}}A_1^*(YX^*Y^{\mathrm{T}} \oplus Z) \\ &\quad = X^*Y^{\mathrm{T}} \oplus X^*Y^{\mathrm{T}}A_1^*A_1 = X^*Y^{\mathrm{T}}(I \oplus A_1^*A_1) \\ &\quad = X^*Y^{\mathrm{T}}A_1^*, \qquad \text{since} \quad I \oplus A_1^*A_1 = A_1^*; \end{aligned}$$

has the lower left block

$$A_1^*YX^*X \oplus A_1^*Y = A_1^*Y(X^*X \oplus I) = A_1^*YX^*;$$

and has the lower right block

$$I \oplus A_1^*YX^*Y^{\mathrm{T}} \oplus A_1^*Z = I \oplus A_1^*(YX^*Y^{\mathrm{T}} \oplus Z) = I \oplus A_1^*A_1 = A_1^*.$$

Compare all this with the blocks of the matrix W of (10). Q.E.D.

ACKNOWLEDGMENTS

The authors thank the referee and Professor Donald B. Johnson for helpful comments and also Sally Goodall for typing this paper.

REFERENCES

1. R. BELLMAN, On a routing problem, *Quart. Appl. Math.* **16** (1958), 87–90.
2. A. BORODIN, J. VON ZUR GATHEN, AND J. HOPCROFT, Fast parallel matrix and GCD computations, *in* "Proceedings, 23rd Annu IEEE FOCS, 1982, pp. 65–71; and *Inform. and Control* **53**, No. 3 (1982), 241–256.

3. R. N. FLOYD, Algorithm 97, shortest path, *Comm. ACM* **5** (1962), 345.
4. G. N. FREDERICSON, "Fast Algorithms for Shortest Paths in Planar Graphs, with Applications," *SIAM J. Comput.* **16**, No. 6 (1987), 1004–1022.
5. H. GAZIT AND G. L. MILLER, An $O(\sqrt{n}\log(n))$ optimal parallel algorithm for a separator for planar graphs, manuscript, Computer Science Dept., University of Southern California, 1986.
5a. H. GAZIT AND G. L. MILLER, A parallel algorithm for finding a separator in planar graphs, *in* "Proceedings, 28th Annu. IEEE Symp. FOCS, 1987, pp. 238–248.
6. M. GONDRAN AND M. MINOUX, "Graphs and Algorithms," Wiley-Interscience, New York, 1984.
7. R. HASSIN, On multicommodity flows in planar graphs, *Networks* **14**, (1985), 225–235.
8. R. HASSIN AND D. B. JOHNSON, An $O(n\log^2 n)$ algorithm for maximum flow in undirected planar networks, *SIAM J. Comput.* **14**, No. 3 (1985), 612–624.
9. D. B. JOHNSON AND S. W. VENKATESAN, Parallel algorithms for minimum cuts and maximum flows in planar networks, *J. ACM* **34**, No. 4 (1987), 950–967.
10. P. KLEIN AND J. REIF, An efficient parallel algorithm for planarity, *in* "Proceedings, 27th Annu. IEEE Symp. FOCS, 1986," pp. 465–477; invited to appear in *J. Comput. System Sci.*
11. E. L. LAWLER, "Combinatorial Optimization: Networks and Matroids," Holt, Rinehart & Winston, New York, 1976.
12. R. J. LIPTON, D. ROSE, AND R. E. TARJAN, Generalized nested dissection, *SIAM J. Numer. Anal.* **16** No. 2 (1979), 346–358.
13. R. J. LIPTON AND R. E. TARJAN, A separator theorem for planar graphs, *SIAM J. Appl. Math.* **36**, No. 2 (1979), 177–189.
14. K. MATSUMOTO T. NISHIZEKI, AND N. SAITO, An efficient algorithm for finding multicommodity flows in planar networks, *SIAM J. Comput.* **14**, No. 2 (1985), 289–302.
15. V. PAN, "How to Multiply Matrices Faster," Lecture Notes in Computer Science, Vol. 179, Springer-Verlag, Berlin 1984.
16. V. PAN AND J. REIF, "Fast and Efficient Solution of Sparse Linear Systems," Technical Report TR-88-16, Computer Science Department, SUNY, Albany, 1988. (Short version *in* "Proceedings 17th Annu. ACM STOC, pp. 143–152. Providence, RI.)
17. V. PAN AND J. REIF, Fast and efficient algorithms for linear programming and for the linear least squares problem, *Comput. Math. Appl.* **12a**, No. 12 (1985), 1217–1227.
18. V. PAN AND J. REIF, "Extension of the Parallel Nested Dissection Algorithm to the Path Algebra Problems," Technical Report 85-9, Comput. Sci. Dept., SUNYA, Albany, NY, 1985.
19. V. PAN AND J. REIF, "Acceleration of Parallel Computations of the Minimum Cost Paths in an Undirected Graph Having a Small Separator Family," Technical Report TR 87-10, Comput. Sci. Dept., SUNYA, Albany, NY, 1987.
20. J. REIF, Minimum s-t cut of a planar undirected network in $O(n\log^2(n))$ time, *SIAM J. Comput.* **12**, No. 1 (1983), 71–81.
21. R. REISCHUCK, A fast probabilistic parallel sorting algorithm, *in* "Proceedings, 22nd Annu. IEEE FOCS, 1981," pp. 212–219.
22. R. E. TARJAN, A unified approach to path problems, *J. Assoc. Comput. Mach.* **28**, No. 3 (1981), 577–593.
23. R. E. TARJAN, Fast algorithms for solving path problems, *J. Assoc. Comput. Mach.* **28**, No. 3 (1981), 594–614.
24. R. C. BACKHOUSE AND B. A. CARRÉ, Regular algebra applied to pathfinding problems, *J. Inst. Math. Applics.* **15** (1975), 161–186.
25. B. A. CARRÉ, An algebra for network routing problems, *J. Inst. Math. Applics.* **7** (1971), 273–294.

Printed by Catherine Press, Ltd., Tempelhof 41, B-8000 Brugge, Belgium