TEL-AVIV UNIVERSITY
RAYMOND AND BEVERLY SACKLER FACULTY OF EXACT SCIENCES
SCHOOL OF COMPUTER SCIENCE

Analysis, Implementation, and Evaluation of Vaidya's Preconditioners

Thesis submitted in partial fulfillment of the requirements for the M.Sc. degree of Tel-Aviv University by

Doron Chen

The research work for this thesis has been carried out at Tel-Aviv University under the direction of Dr. Sivan Toledo

February 2001

Abstract

A decade ago Pravin Vaidya proposed a new class of preconditioners and a new technique for analyzing preconditioners. Preconditioners are essentially easy-to-compute approximate inverses of matrices that are used to speed up iterative linear solvers. Vaidya proposed several families of preconditioners. The simplest one is based on maximum spanning trees (MST) of the underlying graph of the matrix. The second one augments the MST with extra edges to speed up convergence. A third, which Vaidya only mentions briefly, is based on a maximum-weight basis (MWB) of the matroid associated with the graph of the matrix. The first two families apply only to M-matrices, the third to diagonally-dominant symmetric matrices.

This thesis investigates Vaidya's preconditioners. We have implemented all of Vaidya's preconditioners. We have conducted a detailed experimental study of the first two families, which have never been implemented or tested before. Whereas the first two families have been analyzed theoretically before, the third was not. We have analyzed theoretically the convergence properties of the third family. We believe that the tools that we have developed could be used to analyze other preconditioners. We have also developed an efficient algorithm for constructing preconditioners from Vaidya's third family.

Our experimental study of Vaidya's first two families show that they are remarkably robust and can outperform incomplete-Cholesky preconditioners. Our test suite includes problems arising from finite-difference discretizations of elliptic PDEs in two and three dimensions. On 2D problems, Vaidya's preconditioners often outperform drop-tolerance incomplete-Cholesky preconditioners with similar amounts of fill and sometimes outperform modified drop-tolerance incomplete preconditioners. Vaidya's preconditioners do not appear to be effective on 3D problems. Vaidya's preconditioners are robust in the sense that they are insensitive to the boundary conditions of the PDE or to the original ordering of the mesh.

We present an analysis of MWB preconditioners and an efficient algorithm for constructing them. The algorithm uses the generic greedy algorithm for finding a maximum-weight basis in a matroid. To use the generic algorithm, one must provide an algorithm that tests independence. We use a sophisticated augmented union-find data structure that allows us to perform the tests quickly. The correctness of our

method relies on the analysis of the structure of the preconditioner's underlying graph.

Contents

Acknowledgements

Thanks to Cleve Ashcraft for helping us resolve problems with SPOOLES, which we used in early experiments not reported here. Thanks to Haim Kaplan for referring us to Tarjan's paper on augmented union-find data structures.

CHAPTER 1

Background

This chapter provides background material for the next two chapters, which describe original research results. This chapter is based on material from [4], [5], [10], [12], and from Sivan Toledo's lecture notes on high-performance computing.

1.1. Iterative Solvers

The term *iterative methods* refers in this thesis to a wide range of techniques that use successive approximations to obtain more accurate solutions to a linear system $Ax = b$ at each step. These techniques are used when a direct method based on a triangular or orthogonal factorization of A would require too much time or memory. We restrict our attention to symmetric positive-definite matrices.

1.1.1. Conjugate Gradient and Krylov-Subspace Methods. Krylov-subspace methods are a family of iterative solvers that find an approximate solution to $Ax = b$ that lies in the subspace

$$\text{span}\{b, Ab, A^2b, \ldots, A^kb\} \ ,$$

where k is the number of iterations that was performed. It turns out that this subspace, called the *Krylov subspace*, often contains a good approximation $\widehat{x}$ to x even for small k. Furthermore, some Krylov-subspace methods can find the vector in the subspace that minimizes the residual $r = A\widehat{x} - b$ in some norm.

The most widely-used Krylov-subspace methods is called *Conjugate-Gradient*. For symmetric positive-definite matrices, the method finds an approximate solution that minimizes the residual r in the A^{-1} norm, that is, minimizes $\|r\|_{A^{-1}} = (r^T A^{-1} r)$ over the Krylov subspace.

In every iteration, the algorithm computes the direction p in which the approximate solution x will move and the amount α of movement in that direction. When x moves α units in the direction of p, then the residual moves α units in the direction $-Ap$. The direction p is computed by adding to the previous iteration's residual some of the previous direction. This direction basically moves x in the direction of the previous residual, except that we make the direction p A-conjugate to the previous search direction, and, in fact, A-conjugate to all previous search directions ($p_k^T A p_j = 0$ if $j \neq k$).

The number of iterations of the Conjugate Gradient method is bounded above by the square root of the spectral condition number

$\kappa(A)$ of A. (The actual number of iterations can be significantly smaller in some cases.) The condition number is the ratio of the extreme eigenvalues of A, $\kappa(A) = \frac{\lambda_{\max}(A)}{\lambda_{\min}(A)}$.

1.2. Preconditioning

The rate at which an iterative method converges depends greatly on the spectrum of the coefficient matrix. Hence, iterative methods usually involve a second matrix that transforms the coefficient matrix into one with a more favorable spectrum. The transformation matrix is called a *preconditioner*. A good preconditioner improves the convergence of the iterative method, sufficient to overcome the extra cost of constructing and applying the preconditioner.

The idea is to find a matrix M that is easy to invert and that approximates A in some sense. Specifically, we want M to cluster the eigenvalues of A, so that convergence becomes more rapid. The preconditioner conjugate-gradient method solves

$$(M^{-\frac{1}{2}}AM^{-\frac{1}{2}})(M^{\frac{1}{2}}x) = M^{-\frac{1}{2}}b$$

This works well when the eigenvalues of $M^{-\frac{1}{2}}AM^{-\frac{1}{2}}$ are more clustered than the eigenvalues of A.

By manipulating the conjugate-gradient algorithm, it is possible to come up with an algorithm that never uses $M^{\frac{1}{2}}$ or $M^{-\frac{1}{2}}$, and in which the only operation in every iteration is the solution of a linear system $Mz = r$.

1.3. Incomplete Factorization Preconditioners

A broad class of preconditioners is based on incomplete factorizations of the coefficient matrix. We call factorization incomplete if during the factorization process certain *fill* elements, zero positions that would be nonzero in an exact factorization, have been dropped. Such a preconditioner is then given in factored form $M = LU$ with L lower and U upper triangular. The effectiveness of the preconditioner depends on how well M approximates A.

1.3.1. Modified Incomplete Factorizations. This factorization forces the preconditioner to have the same row-sums as the original matrix. Barret et al. explain that in fluid mechanics this has a physical justification [**4**, Section 3.4.2]. More importantly, it has been shown that when A represents a regular 5-point discretization of Laplace's equation with Dirichlet boundary conditions, a no-fill modified incomplete factorization reduced the condition number from $O(n)$ to $O(\sqrt{n})$ (see [**4, 6, 16**] and their reference). An unmodified factorization reduces the condition number by a constant factor, but not asymptotically [**4, 16**].

1.4. Matrices And Graphs

We normally think of matrices as two-dimensional arrays of numbers or as representations of linear transformations. We can also represent matrices as graphs.

DEFINITION 1.4.1. The *underlying graph* $G_A = (V_A, E_A)$ of an n-by-n symmetric matrix A is a weighted undirected graph whose vertex set is $V_A = \{1, 2, .., n\}$ and whose edge set is $E_A = \{(i, j) : i \neq j, a_{ij} \neq 0\}$. The weight of an edge (i, j) is a_{ij}. The weight of a vertex i is the sum of elements in row i of A.

Although the underlying graph is just another representation of a matrix, sometimes it is convenient to analyze certain marix operations as operations on the matrix's underlying graph. For instance, when an unknown i is eliminated during the factoring of a sparse matrix A, the structure of the trailing matrix changes (the remaining equations and variables). The elimination adds edges to the underlying graph between each pair (j, k), where j and k were *neighbors* of i in the graph (edge (j, k) is added if it did not exist before that step).

1.5. Reordering Matrices for Sparsity

When A is factored into triangular factors, the factors usually fill (they are denser than A). Since PAP^T is also symmetric and positive definite for any permutation matrix P, we can instead solve the reordered system $(PAP^T)(Px) = Pb$.

The choice of P can have a dramatic effect on the amount of fill that occurs during the factorization. Thus, it is standard practice to reorder the rows and columns of the matrix before performing the factorization. Reducing fill reduces the amount of memory that the factorization uses and the number of floating-point operations that it performs.

There is no efficient algorithm for finding an optimal ordering. This problem has been shown by Rose and Tarjan to be NP-complete[**22**]. There are, however, several classes of algorithms that work well in practice, like minimum-degree orderings and vertex-separator-based orderings.

Minimum degree is a greedy heuristic which, at each elimination step, eliminates the vertex with the fewest uneliminated neighbors, that is the vertex with the minimum degree. In the matrix, this corresponds to the row and column with the fewest entries in the updated trailing submatrix.

The minimum-degree heuristic is greedy in the sense that it makes reasonable local choices, but without considering their global impact. The heuristic does not even minimize the fill that is generated at every step. After the elimination of the chosen vertex, there will be edges between all its neighbors, but some of these edges might have existed

before this elimination step. Minimum degree minimizes the potential fill at every step, not the actual fill, which is more expensive to compute.

Another important family of ordering algorithms is based on vertex separators, sometimes referred to as a *nested dissection* of the graph. The basic idea is simple. We find a small subset of the vertices of the graph whose removal breaks the graph into at least two connected components, such that no component is very large. We call the small set of vertices a *separator* and we call each connected component a *domain*. We order the vertices of each domain consecutively, one domain after another, and we order the vertices of the separator last.

The vertices within a domain are typically ordered using the same algorithm recursively, by finding a separator which breaks the domain into subdomains and ordering the separator last.

Dissection orderings use a more global view of the elimination process compared to the minimum-degree orderings. Ordering the separator last ensures that no fill edges is created between a vertex v in one domain and a vertex u in another domain. The effect of ordering the separator last, therefore, is to ensure that an entire block of zeros in the original matrix does not fill. By requiring that no domain is very large (does not contain more than $\frac{2}{3}$ of the number of vertices in the graph, for example), we ensure that the zero blocks are large.

The amount of fill and work in a nested-dissection factorization can sometimes be theoretically analyzed. For example, if the graph of an n-by-n matrix A is planar, it has a size $O(\sqrt{n})$ balanced vertex separator. As a consequence, its factors have $O(n \lg n)$ nonzeros and it can be factored in $O(n^{1.5})$ (assuming it has been reordered using nested dissection).

For small matrices, minimum-degree orderings are often more effective than nested dissection. Therefore, modern ordering codes usually combine both dissection and minimum-degree algorithms.

1.6. Discretization of PDEs

In this section we show how discretizations of partial differential equations can produce a system of linear equations, which could be represented as $Ax = b$ where A is an M-matrix.

1.6.1. The Finite-Difference Method for a One-Dimensional Boundary-Value Problem. Let $\zeta^m(I)$ denote the vector space of real functions which are m times continuously differentiable over an interval $I \subset \Re$, m being a non-negative integer. Moreover, f', f'' and $f^{(n)}$ for $n \geq 3$ will denote the successive derivatives of a function of a single real variable.

Consider the following problem. Given the functions c and $f \in \zeta^0([0,1])$ and two constants α and β , find a function $u \in \zeta^2([0,1])$ satisfying

$$-u''(x) + c(x)u(x) = f(x), \;\; 0 < x < 1$$

$$u(0) = \alpha, \;\; u(1) = \beta.$$

Such a problem is called a *boundary-value problem*, because the unknown function has to satisfy the *boundary conditions* $u(0) = \alpha$, $u(1) = \beta$ at the ends of the open interval over which the differential equation has to be satisfied.

If the function c is non-negative over the interval $[0, 1]$, then it can be shown that the problem has a unique solution, which will be denoted by φ.

Except for the rarest of cases, there is no known formula for calculating directly the value of the solution at a general point of the interval $[0, 1]$. As a result, there arises the problem of finding a way of *approximating the values of the solution as closely as may be desired.* One method of achieving this is *the finite-difference method* which will now be described.

Given an integer $N \leq 1$, we set $h = \frac{1}{N+1}$. Let us define a *uniform mesh* of *mesh-size* h over the interval $[0, 1]$ as the set of points $x_i = ih$, $0 \leq i \leq N + 1$ (observe that $x_0 = 0$, $x_{N+1} = 1$), called the *nodes* of the mesh. The mesh-size h is meant to tend to zero. The finite-difference method is a way of obtaining an approximation of the solution φ at the nodes of the mesh, In other words, we look for a vector

$$u_h = \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_N \end{pmatrix} \in \Re^N,$$

such that u_i is 'close' to $\varphi(x_i)$ for $i = 1, 2, \ldots, N$ (the values $\varphi(x_0) = \alpha$ and $\varphi(x_{N+1}) = \beta$ being known), with the accuracy of the approximation improving as h diminishes.

Suppose that the solution φ is four times continuously differentiable in the interval $[0, 1]$. By Taylor's formula, for $i = 1, 2, \ldots, N$, we can write

$$\varphi(x_{i+1}) = \varphi(x_i) + h\varphi'(x_i) + \frac{h^2}{2}\varphi''(x_i) + \frac{h^3}{6}\varphi^{(3)}(x_i) + \frac{h^4}{24}\varphi^{(4)}(x_i + \theta_i^+ h),$$

$$\varphi(x_{i-1}) = \varphi(x_i) = -h\varphi'(x_i) + \frac{h^2}{2}\varphi''(x_i) - \frac{h^3}{6}\varphi^{(3)}(x_i) + \frac{h^4}{24}\varphi^{(4)}(x_i + \theta_i^- h),$$

with $-1 < \theta_i^- < 0 < \theta_i^+ < 1$, so that

$$-\varphi(x_{i+1}) + 2\varphi(x_i) - \varphi(x_{i-1}) = -h^2\varphi''(x_i) - \frac{h^2}{24}\{\varphi^{(4)}(x_i + \theta_i^+ h) + \varphi^{(4)}(x_i + \theta_i^- h)\}.$$

By the intermediate value theorem,

$$\varphi^{(4)}(x_i + \theta_i^+ h) + \varphi^{(4)}(x_i + \theta_i^- h) = 2\varphi^{(4)}(x_i + \theta_i h),$$

with $|\theta_i| \leq \max\{\theta_i^+, -\theta_i^-\} < 1$, so that finally

$$-\varphi''(x_i) = \frac{-\varphi(x_{i+1}) + 2\varphi(x_i) - \varphi(x_{i-1})}{h^2} + \frac{h^2}{12}\varphi^{(4)}(x_i + \theta_i h),$$

with $|\theta_i| < 1$, $1 \leq i \leq N$.

To make the notation less cumbersome, we write

$$\varphi_i = \varphi(x_i), \;\; c_i = c(x_i) \;\; f_i = f(x_i), \;\; 1 \leq i \leq N\,.$$

If we replace the values $-\varphi''(x_i)$ by the expressions given above and taking account of the boundary conditions, we find

$$-\frac{\alpha}{h^2} + \frac{2\varphi_1 - \varphi_2}{h^2} + c_1\varphi_1 = f_1 - \frac{h^2}{12}\varphi^{(4)}(x_1 + \theta_1 h)$$

$$\frac{-\varphi_{i-1}2\varphi_i - \varphi_{i+1}}{h^2} + c_i\varphi_i = f_i - \frac{h^2}{12}\varphi^{(4)}(x_i + \theta_i h), \quad 2 \leq i \leq N-1$$

$$\frac{-\varphi_{N-1} + 2\varphi_N}{h^2} - \frac{\beta}{h^2} + c_N\varphi_N = f_N - \frac{h^2}{12}\varphi^{(4)}(x_N + \theta_N h).$$

The system of equations given above may be written in the matrix form

$$A_h\varphi_h = b_h + \epsilon_h(\varphi),$$

by setting

$$A_h = \begin{pmatrix} 2 + c_1h^2 & -1 & & & & \\ -1 & 2 + c_2h^2 & -1 & & & \\ & \ddots & \ddots & \ddots & & \\ & & -1 & 2 + c_{N-1}h^2 & -1 & \\ & & & -1 & 2 + c_Nh^2 & \end{pmatrix},$$

$$\varphi_h = \begin{pmatrix} \varphi_1 \\ \varphi_2 \\ \vdots \\ \varphi_{N-1} \\ \varphi_N \end{pmatrix}, \quad b_h = \begin{pmatrix} f_1 + \alpha/h^2 \\ f_2 \\ \vdots \\ f_{N-1} \\ f_N + \beta/h^2 \end{pmatrix}, \quad \epsilon_h(\varphi) = \begin{pmatrix} \varphi^{(4)}(x_1 + \theta_1 h) \\ \varphi^{(4)}(x_2 + \theta_2 h) \\ \vdots \\ \varphi^{(4)}(x_{N-1} + \theta_{N-1} h) \\ \varphi^{(4)}(x_N + \theta_N h) \end{pmatrix}.$$

The method relies upon the following heuristic observation. As the mesh-size becomes finer, the vector $\epsilon_h(\varphi)$ becomes 'smaller' (because of the factor h^2). Therefore, one is naturally led to *neglect* it and to

define the following *discrete problem*, associated with the boundary-value problem corresponding to mesh-size h: *find a vector* $u_k \in \Re^N$ *which is a solution of the matrix equation*

$$A_h u_h = b_h.$$

It can be shown that if function c is non-negative, then the discrete problem $A_h u_h = b_h$ has only one solution, and the method converges as the mesh-size h tends to zero, that is, the vector $u_h - \varphi_h$ tends to zero. If function c is non-negative, then the matrix A_h is an M-matrix (symmetric, diagonally dominant with nonpositive off-diagonals).

1.6.2. The Finite-Difference Method for a Two-Dimensional Boundary-Value Problem. The partial differential equation $-\Delta u(x) = f(x)$ for $x \in \Omega$, where $\Delta = \frac{\partial^2}{\partial x_1^2} + \frac{\partial^2}{\partial x_2^2}$, is called *Poisson's equation*, or *Laplace's equation* if $f = 0$. The operator Δ is called the *Laplacian*. Let Γ denote the boundary of Ω. Our problem is that of finding a function $u : \overline{\Omega} \to \Re$ which is the solution of

$$\begin{cases} -\Delta u(x) = f(x) & x \in \Omega \\ u(x) = g(x) & x \in \Gamma \end{cases}$$

The unknown function has to satisfy the *boundary condition* $u = g$ on Γ.

It can be proved that, if the data f and g, as well as the boundary Γ, are sufficiently smooth, then the problem has a unique solution, continuous in $\overline{\Omega}$ and twice continuously differentiable in Ω. Let φ denote the solution.

As in the one-dimensional case, the *finite-difference method* provides an approximation of the solution at a *finite* number of points of the open region Ω. To obtain it, we first set up a *uniform mesh* of the plane, the nodes being the points (ih, jh), $i, j \in Z$. Let Ω_h be the set of nodes of the mesh belonging to Ω, and let Γ_h be the set of points of the plane which fall on the intersection of the boundary Γ and the horizontal and/or vertical line of the mesh.

The discrete problem consists of finding an *approximation of the solution at the points of* Ω_h. As a first step, the points are numbered from left to right and from top to bottom. Although a-priori it may seem that the order in which these points are numbered is of no particular significance, in fact it plays a fundamental role in the practical solution of the associated discrete problem. The first step of the method consists of obtaining an approximate expression for the Laplacian at the nodes $P \in \Omega_h$ by replacing the partial derivatives with appropriate 'finite-difference' quotients. Let P be a point of Ω_h, and let P_i, $1 \leq i \leq 4$, be the four points of the set $\Omega_h \cup \Gamma_h$ which are its closest neighbors in the four directions. If the four points have the same

distance from P, then we can write

$$\Delta_h u(P) = \frac{u(P_1) + u(P_2) + u(P_3) + u(P_4) - 4u(P)}{h^2}.$$

The formula given above is often called the *five-point approximation* of the Laplacian operator. Once the Laplacian approximation has been defined, the *discrete problem*, associated with the boundary-value problem being considered and the chosen mesh, consists of finding a function u_h defined over the discrete set $\Omega_h \cup \Gamma_h$ such that

$$\begin{cases} -\Delta_h u_h(P) = f(P) & P \in \Omega_h \\ u_h(P) = g(P) & P \in \Gamma_h \end{cases}$$

After choosing a *numbering* of the nodes of Ω_h, the two previous relations may be written as a *linear system*

$$A_h u_h = b_h \,,$$

the vector u_h having as its components the values $u_h(P)$, $P \in \Omega_h$.

Notice that the matrix A_h is a sparse M-matrix.

1.7. Support-Graph Theory

This section describes the basic linear-algebra tools that Gremban et al. [**14, 15**], and Bern et al. [**5**] developed to analyze Vaidya's preconditioners. These preconditioners are for M-matrices, i.e. matrices which are symmetric, diagonally dominant (the sum of each row is non-negative), whose off-diagonals are all nonpositive.

When a preconditioner B is used in the Conjugate Gradient method, the number of iterations is proportional to the square root of the ratio between the extreme finite generalized eigenvalues of the pair (A, B), defined below.

DEFINITION 1.7.1. The number λ is a finite generalized eigenvalue of the matrix pencil (A, B) if there exists a vector x such that $Ax = \lambda Bx$ and $Bx \neq 0$. We denote the set of finite generalized eigenvalues by $\lambda_f(A, B)$.

Henceforth, whenever we refer to an eigenvalue of a matrix pencil, we mean a finite generalized eigenvalue.

To bound the amount of work in the Preconditioned Conjugate Gradient method, we need to bound the the finite eigenvalues of (A, B). We need to prove two bounds: an upper bound on $\max \lambda_f(A, B)$ and a lower bound on $\min \lambda_f(A, B)$. We will prove the upper bound directly and the lower bound by proving an upper bound on $\max \lambda_f(B, A) = \frac{1}{\min \lambda_f(A,B)}$.

The main tool that we use to bound $\lambda_f(A, B)$ is the so-called *support* of (A, B), which is the smallest number τ such that $\tau B - A$ is positive semidefinite, $\sigma(A, B) = \min\{\tau : \tau B - A \text{ is positive semidefinite }\}$.

If there is no τ for which $\tau B - A$ is positive semidefinite, we take $\sigma(A, B) = \infty$.

LEMMA 1.7.2. *(Support Lemma* [**15**, Lemma 4.4]*): If* $\lambda \in \lambda_f(A, B)$ *where* B *is positive semidefinite and* $(A) \subseteq (B)$*, then* $\lambda \leq \sigma(A, B)$.

We use this lemma to find upper bound for $\max \lambda_f(A, B)$. Whenever we find a value α such that $\alpha B - A$ is positive semidefinite, we conclude that $\alpha \geq \sigma(A, B)$, and therefore $\alpha \geq \lambda_f(A, B)$.

One way to prove that a matrix is positive semidefinite is to split it into a sum of matrices and prove that each term is positive semidefinite.

LEMMA 1.7.3. *(Splitting Lemma* [**15**, Lemma 4.7]*): Let* $Q = Q_1 + Q_2 + ... + Q_m$*, where* $Q_1, Q_2, ...Q_m$ *are all positive semidefinite. Then* Q *is positive semidefinite.*

The analysis of Vaidya's preconditioners relies on a specific splitting, where the graph of A is split into individual edges and the graph of B into paths. Let G_A be the undirected weighted graph underlying $-A$ and G_B the graph underlying $-B$. Bern et al. show that without loss of generality, both A and B have zero row sums. Therefore the graph structure and edge weights determine the matrices exactly, since all the vertex weights are 0. If the off-diagonal elements of A and B are all negative, then the edge weights in G_A and G_B are positive. Vaidya and Gremban & Miller interpret such graphs as resistive networks where the edge weight is the conductance of a resistor. They split $\tau B - A$ into $(\tau B_1 - A_1) + (\tau B_2 - A_2) + ... + (\tau B_m - A_m)$ such that each A_i corresponds to exactly one edge in G_A and each B_i corresponds to a simple path in G_B. Both the A_i's and the B_i's have nonpositive off-diagonals and zero row sums. Each A_i represents the entire weight of one edge, and each corresponding B_i represents a path that can contain fractions of edge weights. The endpoints of the path represented by B_i are the endpoints of the edge represented by A_i. We will define the *congestion* of an edge in A as the ratio between its weight and the weight of the minimal weight of an edge in the corresponding path in B. The *dilation* of an edge in A will be defined as the length of the corresponding path in B.

LEMMA 1.7.4. *(Congestion-Dilation lemma): Let* G_A *be the underlying graph of a symmetric matrix . Let* $\{A_e\}$ *be the matrices representing each edge in* A *such that* $A = \sum_e A_e$*. Let* $\{B_e\}$ *be the matrices representing paths which support the* A_e*'s (the weights of the edges in the* B_e*'s may be a fraction of the edges' weights in* G_A*). Let* $B = \sum_e B_e$*, where* G_B *is a subgraph of* G_A *(each edge in* G_B *has the same weight as the corresponding edge in* G_A*). Let* α_e *be the congestion of the path supporting edge* i*, and let* β_e *its dilation. Then* $max_e\{\alpha_e \beta_e\}B - A$ *is positive-definite.*

This Lemma gives a bound of $max\{\alpha_e \beta_e\}$ on $\sigma(A, B)$.

CHAPTER 2

Implementation and Evaluation of Vaidya's Preconditioners for M-Matrices

2.1. Introduction

A decade ago Pravin Vaidya proposed an intriguing family of preconditioners for M-matrices [**24**]. He presented his ideas in a scientific meeting but never published a paper on the topic. His preconditioners were never implemented or tested experimentally. We have implemented Vaidya's preconditioners. We experimentally compare the effectiveness of Vaidya's preconditioners to that of incomplete-factorization preconditioners, including no-fill and drop-tolerance preconditioners, both modified and unmodified. Our results indicate that Vaidya's preconditioners are often superior to incomplete-factorization preconditioners.

Vaidya proposed constructing a preconditioner M for a symmetric M-matrix A by dropping off-diagonal nonzeros from A and factoring M (completely). John Gilbert coined the term *complete factorization of incomplete matrices* to describe such preconditioners, as opposed to conventional incomplete factorizations of the complete matrix. Vaidya proposed a sophisticated dropping algorithm that balances the amount of fill in M with the condition number of the preconditioned matrix $M^{-1/2}AM^{-1/2}$. Perhaps the most remarkable aspect of Vaidya's preconditioners is that for many classes of M-matrices, the condition number of the preconditioned matrix depends only on the size of A and is independent of the condition number $\kappa(A)$ of A.

Vaidya's preconditioners work on sparse M-matrices (symmetric diagonally-dominant matrices with nonpositive off-diagonals). Given a parameter t, the method works by constructing a maximum spanning tree T of G_A and splitting T into roughly t connected components of roughly the same size. The method then adds the heaviest edge in G_A between every pair of subtrees if there is an edge between them in G_A. If there are ties and one of the candidate edges is in T, then this edge is added.

A large value for the parameter t results in a small condition number, and hence convergence in a small number of iterations, at the expense of significant fill in the factors of M. (In particular, $t = n$, where n is the dimension of A, leads to $M = A$.) Fill in the factors slows down the factorization of M and slows down each iteration. A

small t on the other hand, yields a higher condition number but sparser factors.

Vaidya stated (without a proof; for a proof, see [**6**]) that for any n-by-n M-matrix with m nonzeros, $t = 1$ yields a condition number $\kappa = O(mn)$. The construction of such a preconditioner costs only $O(m + n \log n)$ work and its factorization costs only $O(m)$ work. For general sparse M-matrices with a constant bound on the number of nonzeros per row, the optimal value of t is around $n^{1/4}$, which yields a total solution time (construction, factorization, and iterations) of $O(n^{1.75})$. For M-matrices whose graphs are planar, the total solution time is only $O(n^{1.2})$ when t is chosen appropriately.

We have evaluated Vaidya's preconditioners by experimentally comparing their performance to that of drop-tolerance incomplete-Cholesky preconditioners (ICC), in the context of a conjugate gradients iterative solver. Both families of preconditioners accept a parameter that indirectly influences the sparsity of the preconditioner: t in the case of Vaidya and the drop tolerance in the case of ICC.

In order to fairly compare the two families of preconditioners, we always compare the performance of preconditioners of equal size, that is, preconditioners that take up roughly the same amount of memory. Keeping the amount of memory fixed means that if one preconditioner fits within main memory so does the other, it implies that preconditioning operations (solving $Mz = r$ in every iteration) take about the same amount of time, and it usually, but not always, leads to similar factorization times. In many of our experiments the factorization times of Vaidya's preconditioners are significantly higher than those of incomplete-factorization preconditioners even when both have similar amounts of fill. We explain this phenomenon in Section 2.3.3.

Our main findings are that Vaidya's preconditioners scale better and are more effective than incomplete-factorization preconditioners on large 2D elliptic problems with Neumann boundary conditions. In particular, on such matrices Vaidya is more effective than drop-tolerance modified incomplete Cholesky preconditioners (MICC). (The unmodified preconditioners were always worse than MICC in our experiments.) Vaidya's preconditioners are also more robust on unisotropic problems than ICC preconditioners. Vaidya's preconditioners are almost unaffected by the boundary conditions, but ICC preconditioners are more effective on Neumann boundary conditions than on Dirichlet. Vaidya's preconditioners converge at similar rates to MICC on Dirichlet problems, but they take longer to construct and factor. Vaidya's preconditioners do not appear in our experiments to be effective on 3D problems. Finally, the scaling behavior of Vaidya's preconditioners on 2D problems shows very slow growth in the number of iterations as the problem grows, slower than that of any of the ICC preconditioners.

The rest of this paper is organized as follows. Section 2.2 describes the theory of Vaidya's preconditioners and our implementation of them. Section 2.3 describes the setup, methodology, and results of our experiments. We summarize our conclusions in Section 2.4.

2.2. Vaidya's Preconditioners

2.2.1. Theory and Construction. In this section we describe the preconditioners that Vaidya proposed and the algorithms that we used to construct them. Vaidya's method constructs a preconditioner M whose underlying graph G_M is a subgraph of G_A. The graph G_M of the preconditioner has the same set of vertices as G_A and a subset of the edges of G_A. (The underlying graph $G_A = (V_A, E_A)$ of an n-by-n symmetric matrix A is a weighted undirected graph whose vertex set is $V_A = \{1, 2, \ldots, n\}$ and whose edge set is $E_A = \{(i, j) : i \neq j \text{ and } A_{i,j} \neq 0\}$; The weight of an edge (i, j) is $-A_{i,j}$.)

The input to the algorithm is an n-by-n M-matrix A, with $2m$ off-diagonal nonzeros and a parameter t. We begin by finding a rooted maximum-weight spanning tree T in G_A. We decompose T into a set of k connected subgraphs $V_1, V_2, \ldots V_k$ such that each V_i has between n/t and $(dn/t)+1$ vertices, where d is the maximal number of children that vertices in T have. We form G_M by adding to T the heaviest edge between V_i and V_j for all i and j. We add nothing if there are no edges between V_i and V_j or if the heaviest edge is already in T. The weight of an edge (i, j) in G_M is the weight of the same edge (i, j) in G_A. We assign weight to a self loop (i, i), which corresponds to a diagonal element $M_{i,i}$, so that the row sums in M and in A are identical. The preconditioner M is the matrix whose underlying graph is G_M.

We denote by M_t the Vaidya preconditioner constructed with the parameter t. We have $M_n = A$. The preconditioner M_1 consists solely of a maximum-weight spanning tree with no added edges. Bern et al. [**6**] show that the condition number of this M_1 is $O(mn)$.

In general, Bern et al. show that the condition number of Vaidya's preconditioner is $O(n^2/k^2)$, where k is the number of subgraphs that T is actually split into. They also analyze the cost of factoring M when G_A is a bounded-degree graph or a planar graph. The results of these analyses are summarized in the Introduction; we omit further details.

2.2.2. Implementation Details. There are two implementation issues that must be addressed in the construction of Vaidya's preconditioners. One is the choice of the maximum-spanning-tree algorithm and the second is the splitting of the tree into subtrees.

We use Prim's algorithm to find the maximum-weight spanning tree T [**21**] because it is fast and because it returns a rooted tree. The root r, which we choose randomly, is an input to Prim's algorithm. (Most textbooks on algorithms describe Prim's algorithm, as well as

$$A = \begin{bmatrix} 12 & -2 & -5 & & -4 & & & \\ -2 & 13 & -5 & & & -6 & & \\ -5 & -5 & 17 & -6 & -1 & & & \\ & & -6 & 8 & & -2 & & \\ -4 & & -1 & & 12 & -3 & -4 & \\ & -6 & & -2 & -3 & 16 & -5 & \\ & & & & -4 & -5 & 15 & -6 \\ & & & & & & -6 & 6 \end{bmatrix}$$

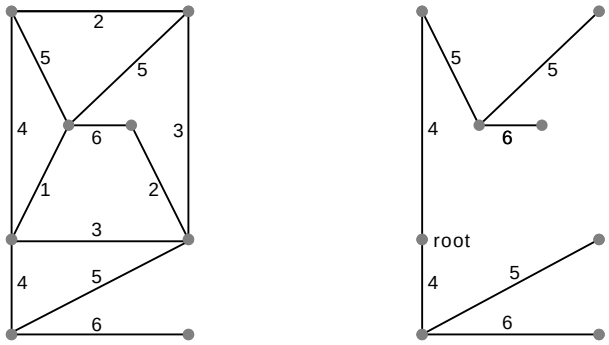


FIGURE 2.2.1. An M-matrix A (top), the graph G_A of A (bottom left), and its rooted maximum spanning tree T (bottom right). The vertices are ordered top to bottom, left to right. The choice of a root for T is arbitrary. Self loops, which correspond to diagonal matrix elements, are not shown.

other maximum spanning tree algorithms; see, for example, [**8**, Chapter 24].) Prim's algorithm returns a rooted tree represented by an array π. The integer $\pi[i]$ represents the parent of the vertex i. We use π to create two length-n integer arrays that allows us to quickly enumerate the children of a vertex. Figure 2.2.1 shows a matrix, its graph and the maximum spanning tree.

We now present a recursive procedure called TREEPARTITION, which is specified in Figure 2.2.2, that decomposes T into a set of connected subtrees. The number of vertices in each subtree is between n/t and $(dn/t)+1$, except for the subtree containing the root, which might be smaller. Ideally, we would have liked to split T into exactly t connected subtrees of nearly equal size, but we are not aware of an algorithm that does so. TREEPARTITION uses a global array s of n integers, where s_i is initialized before the first call to TREEPARTITION to be the number of vertices in the subtree rooted at i. The initial call to TREEPARTITION passes the root of T as an argument.

THEOREM 2.2.1. TREEPARTITION(i) *splits the subtree rooted at* i *into connected subtrees whose size is between* n/t *and* $(dn/t)+1$*, except*

```
TreePartition(vertex i)
    # comment: s_i = number of vertices in the subtree rooted at i
    s_i ← 1
    for each child j of i
        if (s_j > n/t + 1)
            TreePartition(j)
        if (s_j ≥ n/t)
            form a new subtree rooted at j
            disconnect j from i
        else
            s_i ← s_i + s_j
```

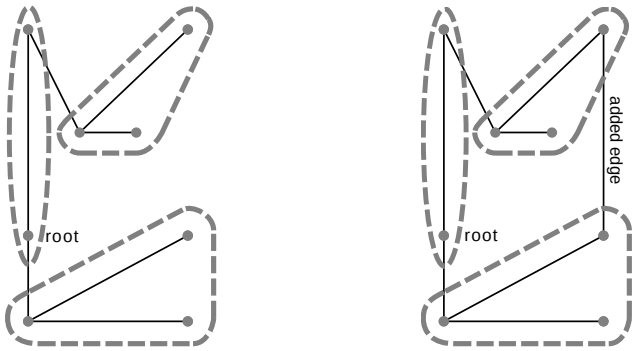
FIGURE 2.2.2. The algorithm that we use to decompose the maximum spanning tree. The code splits the tree T, which is stored in a global data structure. The code uses a global integer array s.

perhaps for the subtree that contains i, which may be smaller (but not larger).

PROOF. We prove by induction on the height of the tree a slightly stronger statement. Namely, that when TREEPARTITION returns, the tree is split appropriately and that the number of vertices in the subtree containing i is s_i. The claim is obviously true for leaves since s_i is set to 1. Suppose that the height of the tree rooted at i is h and that the claim is true for $h' < h$. Clearly, the height of the trees rooted at a child j of i is smaller than h. For each child j, if we call TREEPARTITION(j), then by induction all the subtrees that are formed inside the recursive call have the right sizes and s_j is set to the size of the subtree that remains rooted at j. Therefore, when we test whether s_j is greater or equal to n/t, s_j is correct and is at most $(dn/t) + 1$. If $s_j \geq n/t$, then the subtree rooted at j has a valid size so we can form a new subtree. Otherwise, it is too small and we leave it connected to i and add its size to s_i. When TREEPARTITION(i) terminates, s_i is therefore correct and at most $(dn/t)+1$. The connectedness of the subtrees follows from a similar inductive claim. □

Making the recursive calls only when $s_j > (dn/t) + 1$ is correct, but our experiments indicate that this modification degrades the preconditioner. It appears that if we make the recursive call only when $s_j > (dn/t) + 1$, the subtrees tend to significantly differ in size. If we make the recursive call only when $s_j > n/t + 1$, then the algorithm tends to generate subtrees whose size is close to the lower bound n/t, so they are more uniform in size. We have no theoretical analysis of this phenomenon.

In addition, making a recursive call whenever the subtree is large enough allows the algorithm to partition the graph into as many as n



$$M = \begin{bmatrix} 10 & & -5 & & -4 & & & \\ & 11 & -5 & & & -6 & & \\ -5 & -5 & 16 & -6 & & & & \\ & & -6 & 6 & & & & \\ -4 & & & & 8 & & -4 & \\ & -6 & & & & 11 & -5 & \\ & & & & -4 & -5 & 15 & -6 \\ & & & & & & -6 & 6 \end{bmatrix}$$

FIGURE 2.2.3. The tree from Figure 2.2.1 partitioned (top left), and the graph G_M of the Vaidya preconditioner (top right). The tree was partitioned using TREEPARTITION with $t = 4$. Two edges that connect the 3 subtrees are already in the tree; to complete the preconditioner we added one more edge, the heaviest between the subtrees that are not connected by a tree edge. The matrix on the bottom is the preconditioner M itself.

subgraphs. On the other hand, making the recursive calls only when $s_j > (dn/t)+1$ typically limits the number of subgraphs that the graph can be partitioned into, which in turn limits the condition number that we can reach. Hence, making a recursive call whenever possible gives us more control over fill and condition number than making the recursive call only when necessary.

Adding the heaviest edge in G_A between every pair of subtrees is trivial; we omit the details.

2.3. Experimental Results

2.3.1. Methodology. Both Vaidya's preconditioners and drop-tolerance incomplete-Cholesky preconditioners accept a parameter that indirectly affects the sparsity of the preconditioner. In Vaidya's preconditioner the parameter t affects the number of subgraphs that the tree is split into, whereas in the incomplete-Cholesky preconditioner the drop-tolerance parameter determines which fill elements are dropped from the factors.

We always compare preconditioners with similar levels of fill. The amount of memory required to represent the factors of the preconditioner is proportional to the number of nonzeros in the factor. In addition, the number of floating-point operations required to apply a preconditioner is about twice the number η_L of nonzeros in the factor L of the preconditioner. Hence, the number of nonzeros is a strong determinant of the running time of each iteration in an iterative solver (parallelism and cache effects also influence the running time but we ignore them in this paper). Finally, computing the factors usually takes more time when they are denser, although the dependence is somewhat complex. The number of operations in a complete symmetric factorization is proportional to the sum of squares of nonzero counts for each column.

Thus, one of the input arguments to our testing program is the number of nonzeros in the factors. The program finds the parameter value that yields roughly the requested fill using a binary search. The binary search stops when the amount of fill is 5% or less away from the desired value. The program performs the iterative solution using this parameter value and reports performance data from this run only. We carry out these binary searches only in order to compare the two families of preconditioners fairly. An end user would simply set a value for the fill-controlling parameter based on some analysis or prior experience.

We make the search for a given level of fill more robust using a simple technique. For a fixed rooted maximum spanning tree T, the fill in the factors of Vaidya's preconditioners is not continuous in the parameter t. This can prevent the program from finding a preconditioner with the desired level of fill. To try to overcome this problem, we choose a random root for the tree in every binary-search step. Using this technique, the program is usually able to find a preconditioner whose fill is within 5% of the desired amount; otherwise, the search stops after 100 iterations and uses the best parameter found.

We measure the amount of fill in the factors of preconditioners in terms of *fill ratios*. We define the fill ratio of a preconditioner $M = LL^T$ to be the ratio between the number of nonzeros in L and the number of nonzeros in the factors of a maximum-spanning-tree Vaidya preconditioner. Since the maximum-spanning-tree preconditioner can be factored with no fill, its factors have exactly $2n-1$ nonzeros. Hence, the fill ratio is the ratio between the fill in L and $2n-1$.

2.3.2. Experimental Setup. The experiments were conducted on a dual-processor 600Mhz Pentium III computer with 2GBytes of main memory. We only use one processor. The computer runs the Linux operating system, which on this machine can actually allocate about 1900MBytes to a process in a single `malloc`. The code is written

in C and was compiled using gcc version 2.95.2 with the options `-O3 -mpentium`.

We use METIS version 4.0 [**17, 18**] to find fill-reducing orderings. We use the procedure `METIS_NodeND` with default options to find the ordering. When the graph of the matrix to be factored is a tree, we use a fast special-purpose ordering code that finds a no-fill ordering. This code is essentially a simplified minimum-degree code. Since our experiments use matrices whose graphs are regular meshes in 2 and 3 dimensions, we also run incomplete Cholesky with the natural ordering of the mesh. We expect that for unstructured meshes, envelope minimizing orderings such as Cuthill-McKee [**9**] or Sloan [**19**] would produce results similar to natural orderings of regular meshes [**11**].

We used both the natural row-by-row ordering of the mesh and METIS's ordering for the incomplete factorization. We have found, however, that the modified factorization always break down when the matrix is prepermuted using METIS's ordering. Hence, we only use the natural ordering for MICC. We use both orderings to unmodified ICC.

We implemented a sparse Cholesky factorization algorithm specifically for this project. The code can perform complete, no-fill incomplete (sometimes known as ICC(0) or ICCG(0) [**20**]), and drop-tolerance incomplete factorization. When performing incomplete factorizations, the code can modify the diagonal to ensure that the row sums of LL^T are equal to those of A [**16**]. The code implements a sparse left-looking algorithm. The code is efficient in the sense that its running time is proportional to the number of floating-point operations that it performs. However, the code is not supernodal and is not blocked in any other way, so its performance is poorer than that of state-of-the-art sparse Cholesky codes. We implemented the factorization code because we were not able to find an existing code that has all the incomplete-factorization options that we needed for this experiment.[1]

The iterative solver that we use is preconditioned conjugate gradients (see, for example, [**3, 13**]).

The matrices that we use for our experimental analysis are discretizations of elliptic PDEs on regular 2- and 3-dimensional meshes.

[1] We have been unable to find a state-of-the-art C- or Fortran-callable sparse Cholesky routine that can perform complete, drop-tolerance incomplete, and modified drop-tolerance factorizations (or an LDL^T factorization). We used our relatively slow factorization code since we wanted to use the same code to factor all the preconditioners. We used SPOOLES [**2**] in early experiments but it does not include any modified factorization routine.

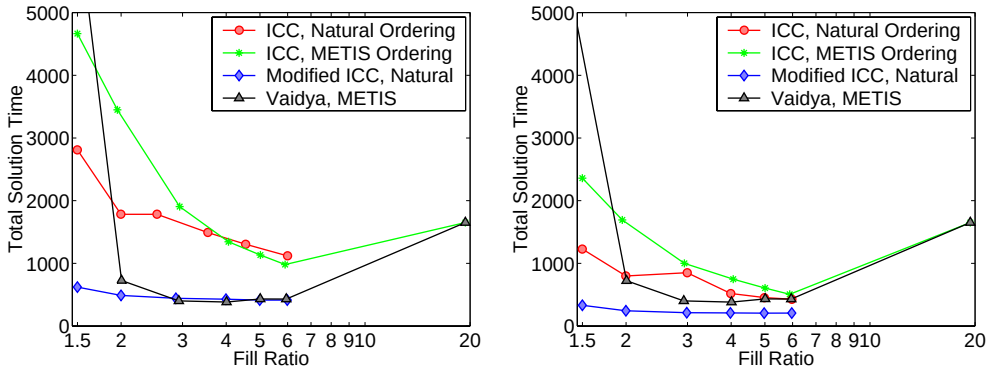


FIGURE 2.3.1. Total solution times with ICC and Vaidya's preconditioners on 1200-by-1200 2D isotropic problems with Neumann (left) and Dirichlet (right) boundary conditions. The graphs show the total solution times as a function of the fill ratio of the preconditioners. The solution times include construction, factorization, ordering, and iterations. The iterative solver stops when the 2-norm of the residual drops by a factor of 10^8. The solution times with a fill-ratio-1 Vaidya's preconditioner are 14539 (Neumann) and 9432 (Dirichlet), outside the scale of the graphs. The rightmost data point in both graphs represents a complete factorization of A with METIS ordering.

The matrices all arise from finite-differences discretizations of the equation

$$c_x \frac{\partial^2 u}{\partial x^2} + c_y \frac{\partial^2 u}{\partial y^2} = f \qquad \text{in } \Omega =]0,1[\times]0,1[$$

with either Dirichlet or Neumann boundary conditions. We solve isotropic problems ($c_x = c_y = 1$) and unisotropic problems in which either $c_x = 100$ and $c_y = 1$ or vice versa. We also solve similar problems in 3D. We use a five-point discretization in 2D and a seven-point discretization in 3D, which lead to a pentadiagonal matrix when a 2D mesh is ordered row-by row (the so-called natural order) or to a septadiagonal matrix in 3D.

We have been unable to find large unstructured matrices in matrix collections, such as MatrixMarket and Tim Davis's collection (over a million unknowns, say).

2.3.3. Experimental Analysis. Both ICC and Vaidya solve problems fastest when the preconditioner is allowed to fill somewhat. If little or no fill is allowed, most of the solution time is spent performing a large number of iterations. If the preconditioner is allowed to fill

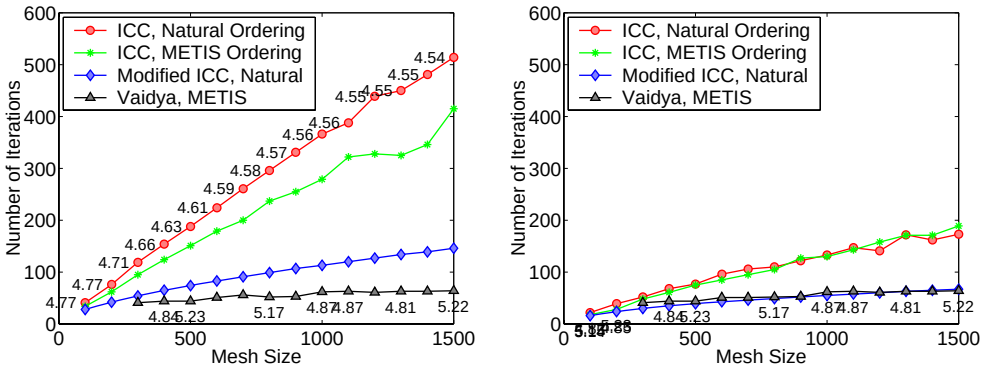


FIGURE 2.3.2. Convergence of fill-ratio-5 ICC and Vaidya's preconditioners on 2D isotropic problems with Neumann (left) and Dirichlet (right) boundary conditions. The graphs show the number of iterations it took to reduce the 2-norm of the residual by a factor of 10^8 as a function of the mesh size $\sqrt{n}$ (i.e., the matrices are n-by-n). The numbers near the graphs show the actual fill ratio of the preconditioners when it deviated from 5 by more than 2.5%.

considerably, most of the solution time is spent factoring the preconditioner. Figure 2.3.1 clearly shows that low and high fill ratios lead to slower solution times than medium fill ratios. The figure also shows that the solution time of the faster preconditioners, Vaidya and MICC, is not sensitive to the fill ratio within the range 3–6. Therefore, we used fill ratio 5 in the rest of the experiments.

In particular, Vaidya's maximum spanning tree preconditioner (with no additional edges) is ineffective. It requires a huge number of iterations, as demonstrated by our experiments and shown in Figures 2.3.1, 2.3.4, and 2.3.5.

Figure 2.3.2 shows that when both Vaidya and ICC preconditioners are allowed to fill appropriately (that is, to achieve near-optimal solution times), Vaidya's preconditioners converge in similar or smaller numbers of iterations. The next best preconditioner in our experiments is always modified ICC with the natural ordering. (We always use METIS to order Vaidya's preconditioners unless their graph is a tree.) Unmodified ICC preconditioners are significantly worse than Vaidya's and MICC.

Vaidya's preconditioners are not sensitive to the boundary conditions that we impose, but ICC preconditioners are. Vaidya's preconditioners converge in roughly the same number of iterations on both Neumann and Dirichlet boundary conditions, but ICC preconditioners are less effective on Neumann boundary conditions. This is shown both by Figure 2.3.2 and by Figure 2.3.4.

TABLE 1. The number of iterations for Vaidya's preconditioners with fill-ratio 5 on isotropic 2D problems with Neumann boundary conditions. The data shows that Vaidya's preconditioners scale well with problem size. The numbers of iterations with Dirichlet boundary conditions are the same, except that 51 rather than 56 iterations were required on grid size 700. The same data are shown graphically in Figure 2.3.2 (but the graphs do not allow close inspection of the scaling behavior).

Grid Size	300	500	700	900	1100	1300	1500
Iterations	41	44	56	53	63	63	64

The scaling behavior of Vaidya's preconditioner is remarkable. Table 1 shows that the number of iterations grows very slowly with the size of the matrix. Note that the size of the preconditioner (i.e., the number of nonzeros in its factors) remains a constant fraction of the size of A. Therefore, these preconditioners combine linear scaling of the work per iteration with a slow growth in the number of iterations, a highly desirable behavior.

Vaidya's preconditioners lead to faster solution times than all ICC preconditioners on Neumann problems, as shown in Figure 2.3.1. On Dirichlet problems, MICC preconditioners are faster, in spite of the fact that Vaidya converges in similar numbers of iterations. This is caused by higher factorization times for Vaidya's preconditioners, which in turn are caused by a less uniform distribution of nonzeros in the columns of the factors. In our experiments, we compare preconditioners with similar fill. That is, we keep the total numbers of nonzeros in the factors similar. The number of floating-point operations in the factorization, however, is proportional to the sum of the squares of the nonzero counts in each column of the factor. Therefore, a preconditioner in which the nonzeros are distributed uniformly among the columns takes less time to factor than a preconditioner with a nonuniform distribution of nonzeros. We have found that Vaidya's preconditioners require 5–10 times more floating-point operations to factor than MICC preconditioners, even when the total fill is similar.

A state-of-the-art factorization code would reduce the solution times with Vaidya's preconditioners more significantly than it would reduce ICC times. Our factorization code is about 5–10 times slower than state-of-the-art codes. Our factorization code runs at about 32Mflops on large sparse matrices, whereas Matlab's sparse Cholesky routine runs at about 190Mflops, about 6 times faster than our code. (Matlab's computational rate was measured on the benchmark machine, with Matlab version 5.3.1; the matrix was generated using `delsq(numgrid('S',802))` and ordered using `symmmd`). A faster factorization code would speed

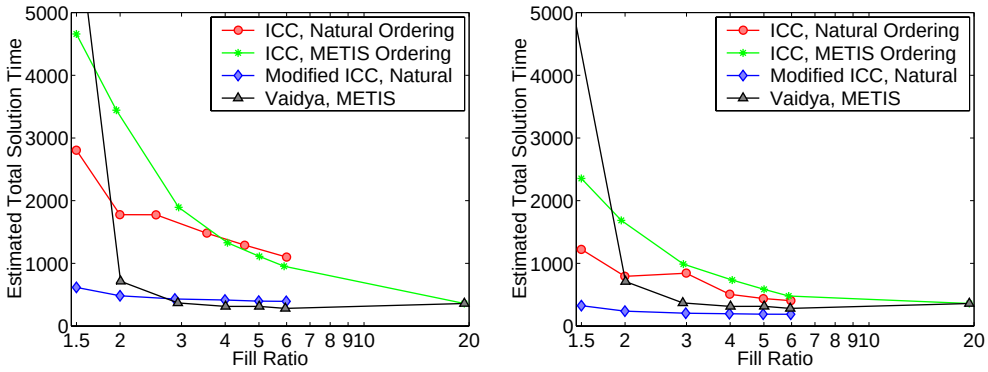


FIGURE 2.3.3. Estimated solution times of ICC and Vaidya's preconditioners on a 1200-by-1200 2D isotropic problems with Neumann (left) and Dirichlet (right) boundary conditions. These graphs show the same data as Figure 2.3.1, but with factorization times scaled down by a factor of 6. These estimates predict the total solution time with a state-of-the art sparse Cholesky factorization code. The estimates suggest that a complete factorization would be slower but quite competitive with an iterative solver, but it uses about 4 times more memory to store the factors.

up all the solvers, but would benefit Vaidya more, since in Vaidya's preconditioners a larger fraction of the total solution time is devoted to the factorization. Figure 2.3.3 estimates solution times with a state-of-the-art sparse Cholesky factorization code. The estimates predict an even greater advantage to Vaidya on Neumann problems. The estimates still predict that MICC would be faster than Vaidya on Dirichlet problems, but the difference would be smaller than our actual measurements indicate.

Vaidya's preconditioners are unaffected by the original ordering of the matrix and are capable of automatically exploiting numerical features. On unisotropic problems, Vaidya's preconditioners are almost completely unaffected by whether the direction of strong influence is the x or the y direction, as shown in Figure 2.3.4. The construction of Vaidya's preconditioners starts with a maximum spanning tree, which always include all the edges (nonzeros) along the direction of strong influence. They are, therefore, capable of exploiting the fact that there is a direction of strong influence and they lead to faster convergence than on an isotropic problem. ICC preconditioners, on the other hand, are sensitive to the interaction between the elimination ordering and the direction of strong influences. Figure 2.3.4 shows that naturally-ordered MICC converges faster for one direction of strong influence than for the

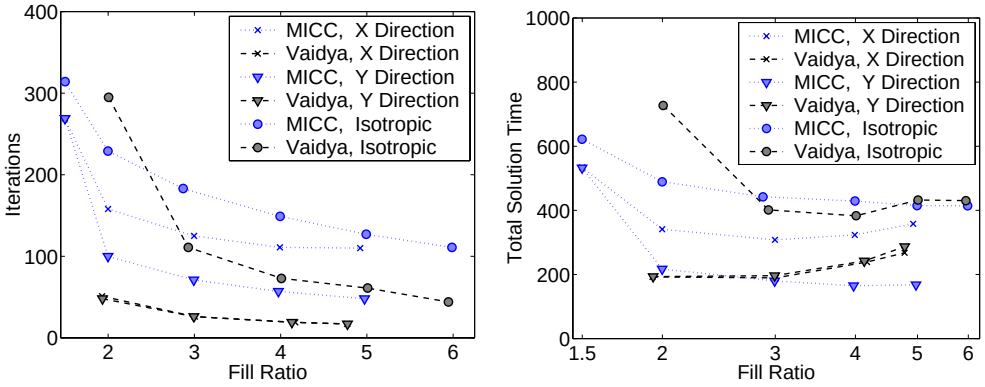


FIGURE 2.3.4. Numbers of iterations (left) and solution times (right) of Vaidya and MICC on unisotropic 1200-by-1200 2D model problems with Neumann boundary conditions. Vaidya's preconditioners are ordered using METIS whereas MICC uses the same natural ordering for all the problems. Vaidya's preconditioners with fill ratio 1 took 2127 seconds (1180 iterations) to converge on the x-direction problems and 1785 seconds (995 iterations) on the y-direction problems.

other when the matrix ordering is fixed. Therefore, to achieve fast convergence with MICC the user would need to tailor the ordering to the numerics, whereas in Vaidya's preconditioners this adaptation occurs automatically.

Vaidya's preconditioners are not as effective as ICC preconditioners on 3D problems, as shown in Figure 2.3.5 for a 100-by-100-by-100 isotropic problem with Neumann boundary conditions. We expect that the difference between Vaidya and MICC would be smaller with a faster factorization code, but Vaidya would certainly remain slower since it converges more slowly. We have experienced similar results with other 3D problems. We note that these results are consistent with the theory of Vaidya's preconditioners. The theory predicts low levels of fill for 2D problems (i.e., for matrices whose graphs are planar), which implies that we can drop a small number of edges from A and obtain a preconditioner with sparse factors. There are no such predictions for 3D problems.

2.4. Conclusions

Vaidya's preconditioners are efficient and robust. On some important classes of matrices, they outperform drop-tolerance incomplete-Cholesky preconditioners (ICC) with similar amounts of fill.

More specifically, we draw the following conclusions from this research:

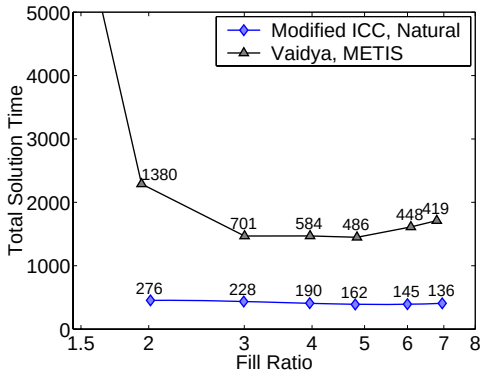


FIGURE 2.3.5. Total solution times as a function of the fill ratio for Vaidya and MICC on a 3D 100-by-100-by-100 isotropic problem with Neumann boundary conditions. The numbers above the data points show the number of iterations. Vaidya uses METIS ordering and MICC uses the natural ordering. Vaidya's running time with fill ratio 1 is 13089 seconds, outside the scale of the graph (fill ratio 1 uses a no-fill ordering).

1. Vaidya's preconditioners are robust when used to solve linear systems arising from finite-differences discretizations of 2D elliptic problems. Their construction and convergence are not affected by the boundary conditions. They effectively and automatically exploit unisotropy. They are not affected by the original ordering of the unknowns. In all of these respects, they are better than all the drop-tolerance preconditioners that we have tested.
2. On some of these 2D problems, Vaidya's preconditioners lead to faster solution times than all the ICC preconditioners, including modified ICC. Specifically, Vaidya's preconditioners are faster on 2D problems with Neumann boundary conditions.
3. Vaidya's preconditioners do not appear to be competitive with ICC preconditions on 3D problems.
4. Vaidya's maximum-spanning-tree preconditioners are sparser than ICC(0) preconditioners, but they converge more slowly. We recommend some fill in Vaidya's preconditioners. In other words, avoid M_1.

Vaidya's preconditioners require high-quality sparse-matrix ordering and factorization codes. The time to construct the preconditioners is typically insignificant, so a parallel ordering, factorization, and triangular solver should enable the application to exploit multiple processors (even if the construction of M is not parallelized).

Our code is publicly available. It includes an iterative solver, the code that constructs Vaidya's preconditioners, the tree-ordering code, a sparse Cholesky factorization code, interfaces to METIS and other

matrix ordering codes, matrix generation and I/O routines. The factorization code can perform complete and drop-tolerance incomplete factorizations, both modified and unmodified. We recommend that users replace our relatively slow factorization code by a faster one.

We recommend that Vaidya's preconditioners be incorporated into software libraries that offer iterative linear solvers, since in some cases they are clearly superior to the widely used incomplete-factorization preconditioners.

We believe that Vaidya's preconditioners are worthy of further investigation. Are there effective Vaidya-like preconditioners for 3D problems? Are there specific maximum spanning trees for problems with constant coefficients that lead to faster convergence? Can we benefit from starting from near-maximal spanning trees with better combinatorial structure? How can we extend Vaidya's preconditioners to more general classes of matrices?

CHAPTER 3

Extending Vaidya's Preconditioners to Symmetric Diagonally-Dominant Matrices

In this chapter we extend Vaidya's maximum-spanning-tree (MST) preconditioners. The new class of preconditioners, which we call *maximum-weight-basis* (MWB) preconditioners, apply to diagonally-dominant symmetric matrices with positive diagonal elements. In contrast, MST preconditioners only apply to M-matrices, which are diagonally dominant with positive diagonals and only negative off-diagonals. In his unpublished manuscript, Vaidya mentions maximum-weight bases, but without any details. The development of the theory and algorithms is original.

It turns out that a recently-proved result by Boman and Hendrickson simplifies the analysis of MWB preconditioners. They show that if $u = Vw$, where u is n-by-1 and V is n-by-k, then $\sigma(uu^T, VV^T) \leq w^T w$. Our strategy is to represent A as a sum of rank-1 matrices $A = \sum_{k=1}^{m} u_k u_k^T$, where each rank-1 matrix correspond to one edge of G_A. We construct a preconditioner $M = VV^T$ by taking V to be a basis for the u_k's that maximizes the trace of $V^T V$. More specifically, we only consider V's whose columns are u_k's.

It turns out that the u_k's define a combinatorial structure known as a matroid, and the problem of finding such a V correspond exactly to the well-studied problem of finding a maximum-weight basis of the matroid. That is why the preconditioners are called maximum-weight-basis preconditioners. It is worth pointing out that if A is an M-matrix, the maximum-weight basis is simply a maximum spanning tree.

This chapter shows that $4mn$ bounds the condition-number of MWB preconditioners. We show the bound in two steps. We bound the lowest eigenvalue with 1 using the fact that A and M are symmetric and diagonally dominant and that G_M is a subgraph of G_A. We bound the highest eigenvalue with $4mn$ using the Boman-Hendrickson lemma, the properties of a maximal-weight basis of a matroid, and a detailed analysis of the structure of G_M.

We also present an efficient algorithm for constructing MWB preconditioners. The algorithm uses the generic greedy algorithm for finding a maximum-weight basis in a matroid. To use the generic algorithm, one must provide an algorithm that tests independence. Although in our case we could use rank-revealing factorizations (say SVDs) to perform these tests, such an implementation would be too expensive.

Instead, we developed a sophisticated data structure that allows us to perform the independence tests quickly. The correctness of our method relies on the analysis of the structure of G_M. Our method is an application of Tarjan's path-compression method [**23**].

3.1. Bounding the Smallest Eigenvalue

This chapter analyzes a certain class of preconditioners for diagonally-dominant symmetric matrices. The graph G_M of the preconditioner M is a subgraph of the graph G_A of A, nonzero off-diagonals in M have the same values as in A, and diagonal elements in M are set up in a way that preserves a generalized row-sum property. For this class of preconditioners, we prove that the smallest eigenvalue of the pencil (A, M) is at least 1. That is, $\lambda_{max}(M, A) \leq 1$.

The preconditioners that we analyze must preserve the generalized row-sums that we define below.

DEFINITION 3.1.1. The *row-weight* of row i of matrix A is $A_{ii} - \sum_{i \neq j} |A_{ij}|$.

We analyze preconditioners whose row weights equal the row weights of A. If the row weights are nonzero, we subtract from both A and M a diagonal matrix D so that the row weights in $A - D$ and $M - D$ are all zero. Clearly, if $(A - D) - (M - D)$ is positive semidefinite, then $A - M$ is positive semidefinite. Thus, we can assume without loss of generality that the row weights in A and M are zero.

The next lemma proves that the small eigenvalue of the pencil is at least 1.

LEMMA 3.1.2. *If A is a diagonally-dominant matrix and M is a preconditioner whose underlying graph is a subgraph of G_A, and whose row-weights are the same as A's, then $\lambda_{max}(M, A) \leq 1$.*

PROOF. The support lemma shows that if $A - M$ is positive semidefinite, then $\lambda_{max}(M, A) \leq 1$. A and M have the same row weights,

$$a_{ii} - \sum_{i \neq j} |a_{ij}| = m_{ii} - \sum_{i \neq j} |m_{ij}|$$

so

$$a_{ii} - m_{ii} = \sum_{i \neq j} |a_{ij}| - \sum_{i \neq j} |m_{ij}| = \sum_{i \neq j} (|a_{ij}| - |m_{ij}|) \ .$$

The matrix M may contain zeros in positions where A contains non-zeros, but all of M's non-zeros are non-zeros in A (with the same values). Since

$$a_{ii} - m_{ii} = \sum_{m_{ij}=0} |a_{ij}| = \sum_{i \neq j} |a_{ij} - m_{ij}| \ ,$$

$A - M$ is diagonally dominant. Its diagonal elements are sums of absolute values, and hence are nonnegative. Such a matrix is positive semidefinite (see, for example,[**3**, Theorem 4.9]). □

We will be able to prove this lemma even more simply, once we prove the Congestion-Dilation lemma for general undirected graphs (lemma 3.3.9). Then we could simply state that each edge in M is supported by the equivalent edge A with congestion 1 and dilation 1.

3.2. The Boman-Hendrickson Lemma and Edge Vectors

We now turn our attention to the largest eigenvalue of (A, M). We propose a preconditioner M whose graph is a specific subgraph of G_A, which allows us to prove an $4nm$ upper bound on the eigenvalues of the pencil. We prove the upper bound using a new tool for bounding support, due to Boman and Hendrickson [**7**]. Their lemma is a generalization of the congestion-dilation lemma.

LEMMA 3.2.1. *(Boman and Hendrickson) if $u \in R^{n\times 1}$ is in the range of $V \in R^{n\times k}$, then $\sigma(uu^T, VV^T) = \min w^T w$ subject to $Vw = u$.*

We use the splitting lemma and Boman and Hendrickson's lemma to prove the upper bound. We split A into a sum of rank-1 matrices $A = \sum_{k=1}^{m} u_k u_k^T$, where each rank-1 matrix correspond to one edge of G_A. We split $4nmM$ trivially into $4nmM = \sum_{k=1}^{m} 4nM$. We then use Boman and Hendrickson's lemma to show that $\sigma(u_k u_k^T, M) \leq 4n$, and hence, that $4nM - u_k u_k^T$ is positive semidefinite. This shows that each of the m terms in the splitting

$$4nmM - A = \sum_{k=1}^{m} \left(4nM - u_k u_k^T\right)$$

is positive semidefinite, and hence the entire sum.

We use Boman and Hendrickson's lemma to show that $\sigma(u_k u_k^T, M) \leq 4n$ by proving that there exist V and w such that $M = VV^T$, $Vw = u_k$, and the entries w_i of w satisfy $|w_i| \leq 2$.

We now show how to represent A as a sum of rank-1 matrices and how to represent M as $M = VV^T$. These representations rely on the following definitions of *edge vectors* and *vertex vectors*.

DEFINITION 3.2.2. The edge vector $\langle ij \rangle$ of a nonzero entry $a_{ij} > 0$ in a matrix A has exactly two non-zeros, $\langle ij \rangle_{\min(i,j)} = 1$ and $\langle ij \rangle_{\max(i,j)} = -1$. The edge vector $\rangle ij \langle$ of a nonzero $a_{ij} < 0$ also has two non-zeros, $\rangle ij \langle_i = 1$ and $\rangle ij \langle_j = 1$. The vertex vector $\langle i \rangle$ of row and column i of a matrix has exactly one nonzero, $\langle i \rangle_i = 1$. All of these vectors are n-by-1 column vectors, where n is the dimension of A.

These vectors can serve as building blocks for symmetric diagonally dominant matrices,

$$\langle ij\rangle\,\langle ij\rangle^T = \begin{pmatrix} \ddots & & & & \\ & 1 & \cdots & -1 & \\ & \vdots & \ddots & \vdots & \\ & -1 & \cdots & 1 & \\ & & & & \ddots \end{pmatrix} \begin{matrix} \vdots \\ \leftarrow \text{ row } i \\ \vdots \\ \leftarrow \text{ row } j \\ \vdots \end{matrix}$$

$$\rangle ij\langle\,\rangle ij\langle^T = \begin{pmatrix} \ddots & & & & \\ & 1 & \cdots & 1 & \\ & \vdots & \ddots & \vdots & \\ & 1 & \cdots & 1 & \\ & & & & \ddots \end{pmatrix} \begin{matrix} \vdots \\ \leftarrow \text{ row } i \\ \vdots \\ \leftarrow \text{ row } j \\ \vdots \end{matrix}$$

The next lemma shows how to represent A as a sum of rank-1 matrices $u_k u_k^T$ where each u_k is an edge vector.

LEMMA 3.2.3. *If A is symmetric and has zero row-weights $a_{ii} = \sum_{i\neq j} |a_{ij}|$, then we can split A into*

$$\begin{aligned} A &= \sum_{\substack{a_{ij}<0 \\ i<j}} |a_{ij}|\,\langle ij\rangle\,\langle ij\rangle^T + \sum_{\substack{a_{ij}>0 \\ i<j}} a_{ij}\,\rangle ij\langle\,\rangle ij\langle^T \\ &= \sum_{\substack{a_{ij}<0 \\ i<j}} \left(\sqrt{|a_{ij}|}\,\langle ij\rangle\right)\left(\sqrt{|a_{ij}|}\,\langle ij\rangle\right)^T + \sum_{\substack{a_{ij}>0 \\ i<j}} \left(\sqrt{a_{ij}}\,\rangle ij\langle\right)\left(\sqrt{a_{ij}}\,\rangle ij\langle\right)^T \end{aligned}$$

PROOF. Each term in the sums contributes to exactly two off-diagonal non-zeros, a_{ij} and a_{ji} and to two diagonal elements a_{ii} and a_{jj}. Furthermore, each off-diagonal nonzero in A receives contributions from exactly one term in the sums. It is easy to see that the contributions sum up to exactly the correct values. □

The preconditioner M is also a sum of rank-1 matrices. The rank-1 matrices whose sum is M are a subset of the rank-1 matrices whose sum is A,

$$M = \sum_{\substack{(i,j)\in E_M \\ a_{ij}<0 \\ i<j}} |a_{ij}|\,\langle ij\rangle\,\langle ij\rangle^T + \sum_{\substack{(i,j)\in E_M \\ a_{ij}>0 \\ i<j}} a_{ij}\,\rangle ij\langle\,\rangle ij\langle^T\ .$$

We define V to be the matrix whose columns are $\sqrt{|a_{ij}|}\,\langle ij\rangle$ and $\sqrt{a_{ij}}\,\rangle ij\langle$ for $i<j$ and $(i,j)\in E_M$. We have $M = VV^T$. The preconditioner M

that we construct satisfies the conditions of the next lemma. Once we show that it does indeed satisfy the conditions, the lemma proves the $4nm$ condition-number upper bound.

LEMMA 3.2.4. *Let $A = UU^T$ and let $M = VV^T$, where U is n-by-m and V consists of the first ℓ columns of U. If for every column u_k of U we have $u_k = Vw_k$ for some w_k with entries whose absolute values are smaller than or equal to 2, then $\sigma(A, M) \leq 4mn$.*

PROOF. We use the splitting lemma to split $A = \sum_{k=1}^{m} u_k u_k^T$ and $mM = \sum_{k=1}^{m} M$ and show that $4nM - u_k u_k^T$ is positive semidefinite. This is true because by Boman and Hendrickson's lemma, $\sigma(u_k u_k^T, M) \leq w_k^T w_k \leq 4n$. □

3.3. The Combinatorial Structure of a Maximum-Weight Basis

Given a set of scaled edge vectors $u_k = \sqrt{|a_{ij}|}\,\langle ij\rangle$ (or $u_k = \sqrt{a_{ij}}\,\rangle ij\langle$) and a weight α_k for each vector u_k, we wish to find a *maximum-weight basis* for the u_k. This basis should consist of a subset of the u_k's and should maximize the sum of the weights of the u_k's in the basis. This section analyses the structure of the maximum-weight basis. We begin by showing a simple property of maximum-weight bases.

LEMMA 3.3.1. *Let $u_1, \ldots, u_\ell$ a maximum-weight basis for the vectors $u_1, \ldots, u_m$ with weights $\alpha_1, \ldots, \alpha_m$ (that is, we assume without loss of generality that the basis consists of the first ℓ vectors). Let $u_k = \beta_1 u_1 + \cdots + \beta_\ell u_\ell$. If $\beta_i \neq 0$ then $\alpha_i \geq \alpha_k$.*

PROOF. Suppose for contradiction that for some i, $\alpha_i < \alpha_k$ and $\beta_i \neq 0$. We show that if we remove u_i from the basis and insert u_k, we end up with another basis with a larger sum of weights. We have

$$u_i = \frac{1}{\beta_i}\left(u_k - \beta_1 u_1 - \cdots - \beta_{i-1} u_{i-1} - \beta_{i+1} u_{i+1} - \cdots - \beta_l u_l\right) .$$

Therefore, the new subset is also spanning. The sum of weights is larger than in the supposedly maximum-weight basis, a contradiction. □

Our next task is more involved. We show that a combinatorial property of a graph ensures that its edge vectors are linearly independent. We need the following definition.

DEFINITION 3.3.2. The *sign of an edge* (i, j) in the graph G_A of a symmetric matrix A is the opposite of the sign of a_{ij}. (That is, the sign is positive if $a_{ij} < 0$.) The *sign of a path* in G_A is negative if it contains an odd number of negative edges; otherwise the path is positive.

We can now state the combinatorial property that guarantees linear independence of edge vectors.

THEOREM 3.3.3. *The edge vectors of an undirected graph G_A are linearly independent if and only if each connected component contains no positive cycles and at most one negative cycle.*

We shall prove the theorem later using three technical lemmas that characterize various ways of spanning an edge vector.

The following lemma shows how to span an edge vector using vectors of edges along a *simple path* between the original edge's endpoints. In this paper we use the term *simple path* to stands for a path in which each *edge* appears only once.

LEMMA 3.3.4. *The edge vectors of a simple positive path between vertices i and j span the edge vector $\langle ij\rangle$. The coefficients of the linear combination are all either 1 or -1. The edge vectors of a simple negative path between vertices i and j span the edge vector $\rangle ij\langle$. The coefficients of the linear combination are all either 1 or -1.*

PROOF. We prove the lemma by induction on the length of the simple path. The claim is clearly true for paths of length 1. Suppose that the lemma is true for paths of length ℓ. Suppose that there is a path of length $\ell+1$ between i and k such that the vertex just before k in the path is j. By induction, the edges of the path from i to j span $\langle ij\rangle$ if that prefix of the path is positive, or $\rangle ij\langle$ otherwise. There are now four cases. If the edge (j,k) is positive and so is the prefix of the path, then either $\langle ij\rangle+\langle jk\rangle$, $\langle ij\rangle-\langle jk\rangle$, $-\langle ij\rangle+\langle jk\rangle$, $-\langle ij\rangle-\langle jk\rangle$ is equal to $\langle ik\rangle$ (the others are $-\langle ik\rangle$, $\langle ik\rangle-2\langle j\rangle$, and $-\langle ik\rangle+2\langle j\rangle$). The second case occurs when (j,k) is positive but the prefix of the path is negative, the third and fourth when (j,k) is negative and the prefix is either positive or negative. Their analysis is similar and is omitted. □

LEMMA 3.3.5. *The edge vectors of a negative cycle that contains vertex i and of a simple path between i and j, where the edges of the path are disjoint from the cycle, span the vertex vector $\langle j\rangle$. The coefficients of the linear combination are ±1 for the edges of the path and $\pm1/2$ for the edges of the cycle.*

PROOF. Let (i,k) be an edge in the cycle. If (i,k) is positive, then the path from i to k along the cycle must be negative, since the entire cycle is negative. Lemma 3.3.4 shows that $\rangle ik\langle$ is a linear combination of the edge vectors along this negative path, with coefficients either 1 or -1. Since $\langle ik\rangle+\rangle ik\langle=2\langle i\rangle$ (if $i<k$; otherwise $-\langle ik\rangle+\rangle ik\langle=2\langle i\rangle$), $\langle i\rangle$ is a linear combination of the edges of the cycle. The coefficients are either $\frac{1}{2}$ or $\pm\frac{1}{2}$. If (i,k) is negative, the rest of the cycle is positive, and a similar argument shows that the cycle spans $\langle i\rangle$. Since the cycle spans $\langle i\rangle$ with coefficients $\pm\frac{1}{2}$ and the path from i to j spans either $\langle ij\rangle$ or $\rangle ij\langle$ with coefficients ±1, the cycle and path together span $\langle j\rangle$ with the desired coefficients. □

LEMMA 3.3.6. *The edge vectors of a connected component that contains a negative cycle span the edge vectors* $\langle ij \rangle$ *and* $\rangle ij \langle$*, for any two vertices* i *and* j *in the component. The coefficients are all* ± 1*,* ± 2 *or* 0*.*

PROOF. Suppose that there is a simple path from i to j that contains cycle edges. Then we can construct another simple path from i to j, in which we will replace the cycle edges in the first path with all the other cycle edges. These two simple paths have opposing signs. Therefore, by lemma 3.3.4, one path spans $\langle ij \rangle$ and the other spans $\rangle ij \langle$, both with coefficients ± 1.

Now suppose that there is no simple path from i to j that contains cycle edges. Let k be the first vertex that is both on the path from i to the cycle and on the path from j to the cycle. Such a vertex must exist, otherwise there is a simple path between i and j that contains cycle edges, a contradiction of our supposition. The vertex k may, however, be one of i and j. The sign of the path between i and j is determined by the sign of the paths between i and k and between k and j. The vectors $\langle ik \rangle$ and $2 \langle k \rangle$ span $\rangle ik \langle$ with coefficients ± 1, which means that the path from i to j and the path from k to the cycle and the cycle span both $\langle ij \rangle$ and $\rangle ij \langle$ with the desired coefficients. □

We are now in position to prove Theorem 3.3.3,

PROOF. ($\Rightarrow$) Suppose to the contrary that there is a positive cycle in G_A. Let e be an edge in that cycle. Then the path between e's endpoints along the cycle has the same sign as e's. Lemma 3.3.4 shows that the vector corresponding to e is a linear combination of the vectors of the edges along the path. Therefore, the vectors are linearly dependent.

Suppose to the contrary that a connected component contains two simple negative cycles. Let us choose a vertex i in the following way: if the two cycles contain common vertices, then we choose i to be one of those vertices. Otherwise we choose i to be one of the vertices on a path connecting the two cycles. Lemma 3.3.5 shows that $\langle i \rangle$ is a linear combination of the vectors corresponding to the edges along any of the paths from i to itself traveling through a negative cycle. Since $\langle i \rangle$ could be represented as two different linear combinations of the edge vectors, the vectors are linearly dependent.

($\Leftarrow$) Let $G = (V, E)$ be a graph, where each connected component contains no positive simple cycles, and at most one negative simple cycle. Suppose to the contrary that the vectors corresponding the edges are linearly dependent. Therefore, there exists a subgraph $G^* = (V, E^*) \subset (V, E)$ and coefficients $\alpha_{ij} \neq 0$, such that

$$\sum_{\substack{(i,j) \in E^* \\ (i,j) \text{ is positive} \\ i<j}} \alpha_{ij} \langle ij \rangle + \sum_{\substack{(i,j) \in E^* \\ (i,j) \text{ is negative} \\ i<j}} \alpha_{ij} \rangle ij \langle = 0 \, .$$

The subgraph G^* cannot contain any leafs. If i is a leaf, only one edge vector contains a nonzero in position i, so this nonzero cannot be canceled out by the other edge vectors in G^*. Also, G^* is a subgraph of G, so each connected component contains no more than one simple cycle. Therefore, G^* is a union of distinct simple negative cycles. By Lemma 3.3.5, each simple negative cycle of length n_0 spans the n_0-dimensional subspace of the n_0 corresponding vertex vectors, and therefore they are linearly independent. No vertex appears in more than one cycle, so the entire set of vectors is linearly independent, a contradiction. □

The characterization of linearly-independent sets of edge vectors that Theorem 3.3.3 will prove useful in the next section, where we use it to efficiently find a maximum-weight basis. Our remaining task in this section is to complete the analysis of the upper bound on the condition number. The next lemma provides the last technical tool that we need.

LEMMA 3.3.7. *Let $u_1, \ldots, u_\ell$ be a maximum-weight basis for a set of m scaled edge vectors $u_k = \sqrt{|a_{ij}|}\,\langle ij \rangle$ (or $u_k = \sqrt{a_{ij}}\,\rangle ij \langle$) with weights $\sqrt{|a_{ij}|}$. Let $u_k = w_1 u_1 + \cdots + w_\ell u_\ell$. Then $\forall 1 \leq r \leq \ell$ $w_r \leq 2$.*

PROOF. Let $u_k = \sqrt{|a_{ij}|}\,\langle ij \rangle$ (or $u_k = \sqrt{a_{ij}}\,\rangle ij \langle$) be the scaled edge vector we want to support. Let $e = (i, j)$ and let G_M be the graph underlying the maximum-weight basis.

We first show how the edge vectors of the edges in the maximum-weight basis support $\langle ij \rangle$ (or $\rangle ij \langle$). This analysis splits into three cases depending on the connected components that i and j belong to. We then show how the maximum-weight basis itself supports u_k.

If i and j are in the same connected component in the maximum-weight basis and that component has no cycles, then the path between i and j must have the same sign as e's, or else e could have been added to the basis. By Lemma 3.3.4, the vector $\langle ij \rangle$ (or $\rangle ij \langle$) is a linear combination of the edge vectors of the edges in the maximum-weight basis with coefficients ± 1 or 0.

If i and j are in the same connected component in the G_M, and that component has a negative cycle, then by Lemma 3.3.6 vector $\langle ij \rangle$ (or $\rangle ij \langle$) is a linear combination of the edges in the maximum-weight basis (without scaling), with coefficients ± 1,± 2 or 0.

If i and j are in two separate connected components, then these two components must both include a negative cycle, or else e could have

been added to the basis. By Lemma 3.3.5, the vector $\langle ij\rangle$ (or $\rangle ij\langle$) is a linear combination of the edges in the maximum-weight basis (without scaling), with coefficients $\pm\frac{1}{2}$, ± 1 or 0.

In all three cases, the $\langle ij\rangle$ (or $\rangle ij\langle$) is a linear combination of the unscaled vectors of the edges in the maximum-weight basis, with coefficients whose absolute values are smaller than or equal to 2. Therefore,

$$w_r = \gamma_r \frac{\sqrt{|a_{ij}|}}{\sqrt{|b_r|}} ,$$

where $\gamma_r \leq 2$ and where the b_r's are the weights of the edges in the MWB. By lemma 3.3.1, $\frac{\sqrt{|a_{ij}|}}{\sqrt{|b_r|}} \leq 1$ for $1 \leq r \leq \ell$. It follows that for $1 \leq r \leq \ell$, $w_r = \gamma_r \frac{\sqrt{|a_{ij}|}}{\sqrt{|b_r|}} \leq 2 \cdot 1 = 2$. □

This concludes the analysis of the condition number of a maximum-weight basis, since we can now apply Lemma 3.2.4 to prove the upper bound on the spectrum. The lower bound has already been established in Lemma 3.1.2. We have, therefore, proven the following theorem.

THEOREM 3.3.8. *The condition-number of a matrix pencil (A, M) where A is symmetric, diagonally dominant with positive diagonals and M is a maximum-weight basis preconditioner is bounded by $4mn$.*

As a side effect of our analysis, we can now formulate and prove an generalized congestion-dilation lemma. We essentially use the same technique that Boman and Hendrickson used to prove the original congestion-dilation lemma [**7**].

LEMMA 3.3.9. *Let $e = (i, j)$ be an edge of weight a. Let u_0 be the scaled vector representing e. Let $V = [u_1, u_2, \ldots, u_\ell]$ be scaled edge vectors $u_k = \sqrt{|b_k|}\,\langle ij\rangle$ (or $u_k = \sqrt{b_k}\,\rangle ij\langle$), corresponding to edges that support e in one of the following ways: either by a simple path whose sign is the same as e's, or by two negative cycles and two paths from each of e's endpoints to the cycles, or by a path from e's endpoints through a negative cycle. Then $\sigma(uu^T, VV^T) \leq \frac{4a}{\min\{b_k\}}\ell$.*

Furthermore, in the first two cases the support $\sigma(uu^T, VV^T)$ is bounded by $\frac{a}{\min\{b_k\}}\ell$.

PROOF. By lemmas 3.3.4, 3.3.5 and 3.3.6, e's vector is a linear combination of the vectors in V, with all the coefficients c_k either ± 2, ± 1 or $\pm\frac{1}{2}$. Let the linear combination coefficients be $(c_1, c_2, \ldots, c_\ell)$. Let $w = (c_1\frac{\sqrt{a}}{\sqrt{b_1}}, c_2\frac{\sqrt{a}}{\sqrt{b_2}}, \ldots, c_\ell\frac{\sqrt{a}}{\sqrt{b_\ell}})^T$. Then $u = Vw$. Therefore:

$$\sigma(uu^T, VV^T) \leq ww^T = \sum_{k=1}^{\ell} c_k^2 \frac{a}{b_k} \leq \sum_{k=1}^{\ell} 4\frac{a}{b_k} \leq \sum_{k=1}^{\ell} \frac{4a}{\min\{b_k\}} = \frac{4a}{\min\{b_k\}}\ell$$

In fact, if the support of u_k is done by one of first two ways, then the coefficients of the linear combination are all either ± 1 or $\pm\frac{1}{2}$, and we have

$$\sigma(uu^T, VV^T) \leq ww^T = \sum_{k=1}^{\ell} c_k^2 \frac{a}{b_k} \leq \sum_{k=1}^{\ell} 1 \cdot \frac{a}{b_k} \leq \sum_{k=1}^{\ell} \frac{a}{\min\{b_k\}} = \frac{a}{\min\{b_k\}}\ell$$

□

As in the analysis of the congestion-dilation lemma for M-matrices, we interpret ℓ to be the dilation and $\frac{a}{\min\{b_k\}}$ to be the congestion.

As we have mentioned, we can use this generalized congestion-dilation lemma to provide another proof of Lemma 3.1.2: each edge in M is supported by the equivalent edge in A with congestion 1 and dilation 1.

3.4. Constructing MWB preconditioners

It turns out that finding a maximum-weight basis for a set of scaled edge vectors is an instance of a well-studied problem. A set of m scaled edge vectors $u_k = \sqrt{|a_{ij}|}\,\langle ij\rangle$ (or $u_k = \sqrt{a_{ij}}\,\rangle ij\langle$) with weights $\sqrt{|a_{ij}|}$ and the collection of linearly-independent subsets define a combinatorial structure called a *matroid*. There exists a generic greedy algorithm for finding a so-called *maximal independent set* in a matroid. In our matroid, a maximal independent set is the maximum-weight basis that we wish to construct.

The generic maximum-weight basis algorithm works by sorting the elements of the matroid (the scaled edge vectors) by weight and trying to add them to the basis, starting from the heaviest. The next vector to be considered is added to the independent set if it is linearly independent of the vectors already in the set.

To apply the generic algorithm, we must provide a routine that tests whether an edge vector is linearly dependent on the vectors already in the set. Using a rank-revealing factorization, such as the singular-value decomposition (SVD) is too expensive. The vectors are highly structured, so we can test for linear independence more efficiently.

The algorithm that we use for testing independence relies on the characterization of independent sets that Theorem 3.3.3 provides. We maintain a data structure that allows us to quickly test whether we can add a new edge vector to the basis. More specifically, we test whether the new edge closes a positive cycle or a second negative cycle in the underlying graph. If so, it is linearly dependent on the edges already in the basis.

The data structure that we use is a forest of shallow rooted trees that represent connected components in the underlying graph. We augment this data structure, which is sometimes referred to as a *union-find* data structure, with labels that allow us to quickly determine

the sign of paths in the graph. The basic union-find data structure was apparently first used by McIlroy and Morris (see [**1**, page 169]); The data structure and its complexity analysis are presented in several textbooks, such as [**1**] and [**8**]. Tarjan proposed the augmentation technique that we use [**23**].

The forest is represented by an array π of length n, where n is the size of the graph. Each rooted tree in the forest represents a connected component of the graph, although the topology of the trees has nothing to do with the topology of the graph. The parent of vertex i is $\pi[i]$. If i is the root of a tree, $\pi[i] = i$. We also maintain an array r; if i is a root then $r[i]$ is the height of the tree rooted at i; r is undefined otherwise. The two arrays π and r are part of the standard implementation of union-find data structure.

Algorithm 1 Finding the representative vertex of a connected component (the root of the tree) with path compression. The algorithm is an augmented version of the standard union-find procedure that also maintains the sign of paths in the graph when the tree is compressed. Indentation denotes block structure.

```
vertex AugmentedFindSet(vertex i)
  temporary vertex j
  if (i ≠ π[i])
      j ←AugmentedFindSet(π[i])
      s[i] ← s[i] xor s[π[i]]
      π[i] ← j
  return π[i]
```

Algorithm 2 Unifies i's and j's trees, using an edge whose sign is ℓ. Returns the root of the united tree.

```
vertex AugmentedUnion(vertex i,vertex j,boolean ℓ)
  temporary vertices ρ_i,ρ_j // representatives of i and j
  ρ_i ← π[i]
  ρ_j ← π[j]
  if (r[ρ_i] > r[ρ_j] )
      π[ρ_j] ← ρ_i
      s[ρ_j] ← s[i] xor s[j] xor ℓ
      return ρ_i
  else
      π[ρ_i] ← ρ_j
      s[ρ_i] ← s[i] xor s[j] xor ℓ
      if (r[ρ_i] = r[ρ_j])
        r[ρ_j] ← r[ρ_j] + 1
      return ρ_j
```

We augment the union-find data structure with two additional bit arrays, s and c. The value $s[i]$ represents the sign of the path in the

graph between i and $\pi[i]$ (0 for positive and 1 for negative); it is only defined if the connected component is a tree. The value $c[i]$ is defined only for roots and specifies whether the connected component has a cycle.

Our algorithm is presented in Algorithms 1, 2, 3 and 4. Algorithm 4 is an instance of the generic greedy maximal-independent-set algorithm, applied to our case. Algorithm 3 tests for independence; it uses Algorithms 1 and 2 as subroutines.

Algorithm 3 Given two vertices i and j and the weight w of the edge connecting them, this algorithm adds (i, j) to the basis if and only if the edge is independent of the current independent set.

```
AddEdgeIfIndependent (vertex i, vertex j, real w)
  temporary vertices ρ_i, ρ_j,unionroot
  temporary boolean edgesign
  edgesign← (w > 0)
  ρ_i ← AugmentedFindSet(i)
  ρ_j ← AugmentedFindSet(j)
  if (ρ_i ≠ ρ_j)
      // i and j are in different connected components
      if ((c[ρ_i] = 0)or(c[ρ_j] = 0))
          // one of the connected components does not contain a
cycle
          AddEdgeToBasis(i, j, w)
          unionroot ← AugmentedUnion(i, j,edgesign)
          c[unionroot] ← c[ρ_i] or c[ρ_j]
  else
      // i and j are in the same connected component
      if ((edgesign ≠ s[i] xor s[j]) and(c[ρ_1]==0))
          // the connected component does not contain a cycle, and
          // adding (i, j) does not close a positive cycle
          AddEdgeToBasis(i, j, w)
          c[ρ_i] ← 1
```

Algorithm 4 This is the generic greedy algorithm to find the maximal-independent set of a matroid.

```
GreedyMaximumWeight()
  Sort(edges by absolute value of weight)
  foreach (e=(i,j) an edge of weight w)
    if (EdgeIsIndependent(i, j, w))
      AddEdgeToBasis(e)
```

The correctness of the algorithm relies on the correctness of the generic greedy algorithm, the correctness of the union-find data structure, on Theorem 3.3.3, and on the correct maintenance of the arrays

s and c. The correct maintenance of c is trivial. The correct maintenance of s is more challenging to prove. We start with a simple technical lemma.

LEMMA 3.4.1. *If i, j, and k are vertices in a connected component of G_A that contains no cycles and*

$$s[i,j] = \begin{cases} 0 & \text{if the path in } G_A \text{ between } i \text{ and } j \text{ is positive} \\ 1 & \text{otherwise,} \end{cases}$$

then $s[i,k] = s[i,j] \operatorname{xor} s[j,k]$.

PROOF. Since the connected component is a tree, the paths from i to j and from k to j must meet at some vertex x; from x to j the two paths are identical (x and j may be the same vertex). The simple path from i to k is in fact concatenation of the path from i to x to the path from x to k. Therefore,

$$\begin{aligned} s[i,j] \operatorname{xor} s[j,k] &= (s[i,x] \operatorname{xor} s[x,j]) \operatorname{xor} (s[j,x] \operatorname{xor} s[x,k]) \\ &= s[i,x] \operatorname{xor} (s[x,j] \operatorname{xor} s[j,x]) \operatorname{xor} s[x,k] \\ &= s[i,x] \operatorname{xor} s[x,k] \\ &= s[i,k] \ . \end{aligned}$$

□

The next three lemmas show that the algorithm does, indeed, maintain s correctly.

LEMMA 3.4.2. AUGMENTEDFINDSET *preserves the correctness of s. That is, if the array s is correct before the call of* AUGMENTEDFINDSET, *then it is correct after the subroutine returns.*

PROOF. AUGMENTEDFINDSET changes the values of π and c along the path from a vertex i to the root. We prove the correctness by induction on the distance from the root. If i is the root, the algorithm returns immediately, so the claim holds. Suppose the lemma is correct for all the vertices between vertex i and the root. By lemma 3.4.1, we have that $s[i, \text{root}] = s[i, \pi[i]] \operatorname{xor} s[\pi[i], \text{root}]$. The parent of $\pi[i]$ the recursive call is the root, and the parent of i when the subroutine returns is also the root, so correctness is maintained. □

LEMMA 3.4.3. *If the arguments i and j to* AUGMENTEDUNION *are immediate children of roots and if s is correct before the call, then s is maintained correctly by* AUGMENTEDUNION.

PROOF. The only change in the array π is $\pi[\rho_j] = \rho_i$ or $\pi[\rho_i] = \rho_j$. By lemma 3.4.1 $s[\rho_i, \rho_j] = s[\rho_i, i] \operatorname{xor} s[i,j] \operatorname{xor} s[j, \rho_j]$. By the hypothesis of the lemma, $\rho_i = \pi[i]$ *and* $\rho_j = \pi[j]$. *Since* $s[\rho_i, i] = s[i]$, $s[j, \rho_j] = s[j]$ *and* $s[i,j] = \ell$, *the lemma is correct.* □

LEMMA 3.4.4. ADDEDGEIFINDEPENDENT *is correct.*

PROOF. If i and j are in different connected components and both contain cycles, the routine returns without adding (i, j) to the basis. If they are in different components and at most one contains a cycle, the routine adds (i, j).

If i and j are in different components, then $s[i] \operatorname{xor} s[j]$ is the sign of the path between them, since both are children of the same root. In that case, the routine adds the edge if and only if the sign of the edge is different from the sign of the path (i.e., the edge closes a negative cycle) and there is no cycle in the component.

The correct maintenance of s follows from the fact that we call AUGMENTEDUNION only when the arguments are children of roots. □

The complexity analysis of the algorithm is simple. It shows that the running time of the algorithm is dominated by sorting the edges. The total cost of the calls to ADDEDGEIFINDEPENDENT is essentially linear in m. The proof is essentially identical to Tarjan's analysis of augmented union-find data structures in **[23]**.

THEOREM 3.4.5. GREEDYMAXIMUMWEIGHT *runs in* $O(m \lg m + m\alpha(m, n))$ *where* α *is the inverse of Ackermann's function.*

PROOF. Sorting the edges takes $O(m \lg m)$ time.

We make m calls to ADDEDGEIFINDEPENDENT. Each call makes two calls to AUGMENTEDFINDSET and at most one to AUGMENTEDUNION. The other costs in ADDEDGEIFINDEPENDENT are $O(1)$. Since AUGMENTEDFINDSET and AUGMENTEDUNION are $O(1)$ modifications to the corresponding standard union-find routines, the total cost of all the calls is $O(m\alpha(m, n))$. □

Bibliography

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[2] Cleve Ashcraft and Roger Grimes. SPOOLES: An object-oriented sparse matrix library. In *Proceedings of the 9th SIAM Conference on Parallel Processing for Scientific Computing*, San-Antonio, Texas, 1999. 10 pages on CD-ROM.

[3] Owe Axelsson. *Iterative Solution Methods*. Cambridge University Press, 1994.

[4] R. Barret, M. Berry, T. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. SIAM, Philadelphia, PA, 1993.

[5] Marshall Bern, John R. Gilbert, Bruce Hendrickson, Nhat Nguyen, and Sivan Toledo. Support-graph preconditioners. In *Proceedings of the Copper Mountain Conference On Iterative Methods*, page 7 unnumbered pages, Copper Mountain, Colorado, 1998.

[6] Marshall Bern, John R. Gilbert, Bruce Hendrickson, Nhat Nguyen, and Sivan Toledo. Support-graph preconditioners. Technical report, School of Computer Science, Tel-Aviv University, 2001.

[7] Eric Boman and Bruce Hendrickson. Support theory for preconditioners. Unpublished manuscript.

[8] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press and McGraw-Hill, 1990.

[9] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 24th National Conference of the Association for Computing Machinery*, pages 157–172, 1969.

[10] James W. Demmel. *Numerical Linear Algebra*. Berkeley Mathematics Lecture Notes, 1993.

[11] Iain S. Duff and Gérard Meurant. The effect of ordering on preconditioned conjugate gradient. *BIT*, 29(4):635–657, 1989.

[12] A. George and J. W. H. Liu. The evolution of the minimum-degree ordering algorithm. *SIAM Review*, 31:1–19, 1989.

[13] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. Johns Hopkins University Press, 3rd edition, 1996.

[14] K.D. Gremban, G.L. Miller, and M. Zagha. Performance evaluation of a parallel preconditioner. In *9th International Parallel Processing Symposium*, pages 65–69, Santa Barbara, April 1995. IEEE.

[15] Keith D. Gremban. *Combinatorial Preconditioners for Sparse, Symmetric, Diagonally Dominant Linear Systems*. PhD thesis, School of Computer Science, Carnegie Mellon University, October 1996. Technical Report CMU-CS-96-123.

[16] I. Gustafsson. A class of first-order factorization methods. *BIT*, 18:142–156, 1978.

[17] George Karypis and Vipin Kumar. Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, 48:96–129, 1998.

[18] George Karypis and Vipin Kumar. A parallel algorithm for multilevel graph partitioning and sparse matrix ordering. *Journal of Parallel and Distributed Computing*, 48:71–85, 1998.

[19] Gary Kumfert and Alex Pothen. Two improved algorithms for reducing the envelope size and wavefront of sparse matrices. *BIT*, 18:559–590, 1997.

[20] J. A. Meijerink and H. A. van der Vorst. An iterative solution method for linear systems of which the coefficient matrix is a symmetric M-matrix. *Mathemathics of Computation*, 31:148–162, 1977.

[21] R. C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36:1389–1401, 1957.

[22] D. J. Rose and R. E. Tarjan. Algorithmic aspects of vertex elimination on directed graphs. *SIAM Journal on Applied Mathematics*, 34:176–197, 1978.

[23] Robert Endre Tarjan. Applications of path compression on balanced trees. 1979.

[24] Pravin M. Vaidya. Solving linear equations with symmetric diagonally dominant matrices by constructing good preconditioners. Unpublished manuscript. A talk based on the manuscript was presented at the IMA Workshop on Graph Theory and Sparse Matrix Computation, October 1991, Minneapolis.