TOPICS IN OPTIMIZATION AND SPARSE LINEAR SYSTEMS

BY

ANIL JOSHI

B.Engr., Osmania University College of Engineering, Hyderabad, 1980
M.S., Polytechnic University of New York, 1990

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1997

Urbana, Illinois

TOPICS IN OPTIMIZATION AND SPARSE LINEAR SYSTEMS

Anil Joshi, Ph.D.
Department of Computer Science
University of Illinois at Urbana-Champaign, 1997
Professor Pravin M. Vaidya, Advisor

Abstract

In the first part of the thesis, we study three problems in optimization - Network Optimization Problem, A Convex Optimization Problem arising out of Timing Analysis of VLSI circuits, and A Geometric Solution to two variable Convex Optimization Problems.

In the second part of the thesis, we analyze the quality of a new graph-based preconditioner for large sparse Symmetric Positive Definite Diagonally Dominant (SPDDD) linear systems. These kinds of linear systems arise in the solution of scalar second order PDEs for Heat Transfer, Electrostatics, Electromagnetics, Ground Water Flow, and Diffusion (with or without reaction) when they are discretized using finite differences. They also arise in discrete problems like Network Flow Problems (Assignment, Maximum Flow, and Minimum Cost Flow), Large Resistive Networks, and Laminar Flow in Pipe Networks.

To
my parents Vasantha Rao and Chandrakala Joshi
my wife Bharathi and our daughter Divya

Acknowledgements

I would like to thank my advisor Pravin for unparalleled advise and guidance. He had been the epitome of patience. I would be very happy indeed if I am able to achieve even a tenth of what he had achieved already. I would also like to thank the members of my committee, Professors C.L. Liu and Faisal Saied for their helpful comments.

I would like to thank my father Vasantha Rao Joshi for kindling my interest in geometry and chess. I owe much of my success to him. My mother and my grand mother Radhabai Joshi have been very supportive and loving over the years. Without their sacrifices, this wouldn't have been possible.

I would like to thank Arthur for taking our relationship all the way from being collaborators on research to very close friends. I have a very special place in my heart for Kamalakar and Jaisree who had engaged me in lively intellectual discussions (some times into the wee hours of the morning) about everything in the world.

I would like to thank Glen Jennings for coming down all the way from Sweden to collaborate on the VLSI timing analysis problem.

I would like to thank Danny, Padmaja and Srinivas, all my Osmania friends in New Jersey - Sudarshanam, Vijay Kumar, Srinivas Raghuram, and Sunder, for their unwavering faith in my ability to succeed in adversity. I thank Rajeeva for being a confidant and a sounding board of my ideas.

I would like to thank my room mates - Prashant, Anmol, and Vivek - for the enjoyable times I had with them. I also would like to thank Nimish, Nataraj and Lata, Nancy, Biju, Gadi, Lun, Tun, Srikanth and Edgar for providing lively company during my Ph.D. years.

During these long years, I have seen my younger brother Ajit come to the United States and grow from a diffident young man to a confident professional. I thank him for making numerous trips to Champaign just to spend time with me. I also would like to thank Aparna and Yoganandam for support when I most needed it. I thank my long time friend Suri for taking time out of his busy schedule and call me up whenever he was in USA on his consulting trips.

I would like to thank Swati and Pravin for inviting me over to their house several times.

I thank my wife Bharathi for patiently waiting in Bombay for over a year for the immigration paperwork to navigate the system. She had provided me with all the support a husband can ask for and more. I thank her for giving me a wonderful present, our daughter Divya, just before Father's Day. Bharathi, I thank you for putting up with my schedule for the last two years and for taking care of Divya single handedly.

Financial Support This work was supported in part by NSF under contracts CCR-9057-481 and CCR-9007-195 and in part by Prof. Vaidya's PYI Various Donors Grant VARDNRS 1-6-41990.

Table of Contents

Chapter

1 Introduction 1

2 Implementation of a Path-Following Algorithm for Network Optimization . . 3
- 2.1 Introduction 3
- 2.2 An Overview of the Path-Following Algorithm 4
- 2.3 Solving the Linear System at Each Step 9
- 2.4 Some Techniques to Improve Speed and Stability 12
- 2.5 Experimental Results and Analysis 15
- 2.6 Conclusions 17

3 Practical Cycle Borrowing Analysis Using Interior Point Algorithms 36
- 3.1 Definitions and Previous Work 37
- 3.2 Generating the Linear Program from the Circuit 40
- 3.3 Analysis Methods 41
- 3.4 Test Circuits and Results 44
- 3.5 Summary and Conclusions 46

4 An Efficient Algorithm for Special Two-Variable Convex Programming Problems 47
- 4.1 Introduction 47
- 4.2 Notations and Definitions 48
- 4.3 Some Theorems 49
- 4.4 A Linear Time Algorithm 50
  - 4.4.1 Psuedocode Description of the Algorithm 51
  - 4.4.2 Detailed explanation of the Algorithm 51
- 4.5 Analysis 53
- 4.6 Discussion 54

5 Analysis of SPDDD Linear Systems Whose Graphs Have Restricted Topologies 55
- 5.1 Motivation 55
- 5.2 Certain SPDDD Matrices and Associated Graphs 56
- 5.3 Problem Description 57
- 5.4 Constructing a Good Preconditioner 59
  - 5.4.1 Approximating SPDDD Matrices 59
  - 5.4.2 Approximations of Grid Graphs 61

5.5 Solution Methodology 69
5.5.1 Overview of the Solution Method 69
5.5.2 Exact and Inexact Chebyshev Iterative Methods 71
5.5.3 Exact Chebyshev Iterative Method 71
5.5.4 Inexact Chebyshev Iterative Method 72
5.6 Two Level Algorithm 75
5.7 Recursive Algorithm 76
5.7.1 Analysis of the Recursive Algorithm 76
5.8 Other Recursive Topologies 79
5.9 Missing Edges and Boundary Conditions 82
5.10 Scope for Further Work 84

6 Conclusions 85

BIBLIOGRAPHY 86

Vita 91

Chapter 1

Introduction

There are five chapters in this thesis. In Chapters 2, 3, and 4 we study three optimization problems. In Chapter 5 we give good analysis for solving sparse Symmetric Positive Definite Diagonally Dominant (SPDDD) linear systems with associated graphs that have special topologies. We also discuss why solving these kinds of linear systems is important and its relevance to the network flow problem described in Chapter 2. These linear systems arise in the solution process of several important continuous as well as discrete physical problems.

In Chapter 2 we describe a fast implementation of an interior point algorithm for solving large network optimization problems such as Assignment, Maximum Flow, and Minimum Cost Flow problems. All these problems can be cast as Linear Programming problems and any of the existing LP algorithms (*e.g.* Simplex, Ford-Fulkerson, and any number of Interior-Point based algorithms) may be employed to solve these problems. We describe an implementation of a path-following algorithm that works well if the problem specifics (*i.e.* these problems are network flow problems rather than general LP problems) are taken into consideration and appropriate data-structures are chosen. We also give asymptotic growth rate comparisons of our implementation with two widely available codes - NETFLO and RELAXT III - on several network flow problems that have been generated automatically by a generator at a workshop on network flows conducted by DIMACS. Because these problems are network flow problems, at every step in the path-following algorithm, we need to solve a sparse Symmetric Positive Definite Diagonally Dominant (SPDDD) Linear System. The asymptotic growth rates of the number of iterations in Preconditioned Conjugate Gradient (PCG) observed in this implementation on several network flow problems have encouraged us to

attempt to analyze the goodness of preconditioners for SPDDD linear systems whose graphs have restricted topologies which we do in Chapter 5.

In Chapter 3, we describe a formulation of the Cycle Borrowing Timing Analysis Problem for large circuits as a convex programming problem. We use the Convex Programming Algorithm described in [44] to solve this problem. An interesting twist in the solution method is that the network flow implementation described in Chapter 2 is used to generate the separating hyperplane required by this algorithm. The algorithm in [44] requires an oracle which generates a separating hyperplane through a test point such that either the plane separates the test point from the feasible region or defines a half space containing the optimum if the test point is feasible. The Cycle Borrowing Timing Analysis Problem is formulated as a Convex Programming problem in a low-dimensional space where the oracle generates a cutting plane by solving the dual of a network flow problem. We give results on a number of large circuits that are used by students of the VLSI architecture course at Lund University. We use the implementation [18] of algorithm [44] in addition to our Fast Path-following Network Flow Algorithm described in Chapter 2.

In Chapter 4, we describe a linear time algorithm for special convex programs with linear objective functions in two-dimensions. This is an extension of the linear time algorithms in [14, 35] for Linear Programming problems in $\Re^2$.

In Chapter 5, we analyze the quality of a new graph-based preconditioner for SPDDD matrices whose graphs have restricted topologies. These kinds of SPDDD matrices arise in the solution of scalar second order Partial Differential Equations (PDEs) (*e.g.* Heat transfer, Electrostatics, Electromagnetics, Ground water flow, Diffusion, etc.). They also occur in discrete problems such as network flows and resistive electrical networks.

Chapter 2

Implementation of a Path-Following Algorithm for Network Optimization

2.1 Introduction

Many network optimization problems - Assignment, Maximum Flow, and Minimum Cost Flow - can be cast as Linear Programming (LP) problems. Recently several algorithms have been proposed for LP which belong to the general class of algorithms known as Interior Point methods (See [46, 45, 38, 32]). The objective of our implementation is to show that path-following interior point algorithms can work well if the problem peculiarities are taken into consideration and appropriate data-structures are chosen. Computational experiments comparing the performance of Network Simplex and Interior Point methods have been carried out by several researchers. The results can be found in [1, 31, 39, 37, 2].

We use the notation e for the number of edges and v for the number of vertices in the given network. A network flow problem is defined as the following LP [16].

$$\max c^T x$$

subject to the lower and upper bound (*i.e.* capacity) constraints

$$l_i \leq x_i \leq u_i$$

and the flow conservation constraints

$$Ax = b$$

where $x \in \Re^e$ is the flow vector whose ith element is the flow in edge i. l_i and u_i are the lower and upper bounds respectively on the flow in edge i. In addition, these capacities are assumed to be integral. Matrix $A \in \Re^{(v-1)\times e}$ describes the network under consideration. A is obtained from the node-arc incidence matrix of the network by dropping the row corresponding to a pre-specified ground node. Each column of A corresponds to an edge of the network. If the edge k is directed from node i to node j then there is a -1 in the ith row and a $+1$ in the jth row in column k of A (If edge k is incident on the ground node there is a $+1$ or a -1 in column k of A). All other entries in that column are zero. Vector $b \in \Re^{v-1}$ corresponds to the node flow values (with appropriate sign for a source or a sink). Vector $c \in \Re^e$ is the cost vector.

Any algorithm for LP can be employed to solve this problem. But since we know that this is a network flow problem, it is possible to speed up the LP algorithm by taking advantage of the special structure of matrix A and the form of the upper and lower bound constraints. This formulation is suitable for Assignment, Maximum Flow and Minimum-Cost Flow problems after some minor transformations are applied to the network, the cost vector c and the node-flow vector b. In Section 2.2 the algorithm used to solve this problem is described (See [46] for a detailed description and rate-of-convergence). One of the time consuming computations in the algorithm is to solve a sparse linear system at each step. We use a Pre-conditioned Conjugate Gradient (PCCG) method to solve the linear system (See [20]). In Section 2.3, the construction of the pre-conditioner is described. Section 2.4 describes additional heuristics such as reduction of problem size by fixing flow values in certain edges, approximate centering, etc. that may be employed to speed up the algorithm in practice. The experimental results are presented in Section 2.5 and analyzed.

2.2 An Overview of the Path-Following Algorithm

The network polytope is defined by the upper and lower bound constraints which are the bounding hyperplanes of the polytope. It is denoted as

$$P = \{x : Ax = b,\ l_i \leq x_i \leq u_i\}.$$

The path-following algorithm employed in this implementation follows the trajectory of analytic centers of a family of polytopes defined by the intersection of the network polytope and the objective function hyperplane, *i.e.* the centers of the family of polytopes $P(\beta)$ defined as:

$$P(\beta) = P \cap \{x : c^T x \geq \beta\}.$$

The analytic center of $P(\beta)$ is the minimizer of the strictly convex logarithmic barrier function $F(x, \beta)$ where

$$F(x, \beta) = -(\sum_{i=1}^{e} (\ln(x_i - l_i) + \ln(u_i - x_i)) + w \ln(c^T x - \beta)).$$

We put a large weight w on the objective function hyperplane so that the analytic center is pushed away from the objective function hyperplane $c^T x = \beta$ towards the optimal facet of the network polytope. Specifically the weight w equals the number of edges e times a large multiplier (which is chosen to be $\min\{\sqrt{e}, \sqrt{100,000}\}$).

Clearly, when the flow in any edge equals either the capacity, the lower bound or is such that $c^T x = \beta$ then this function goes to ∞. At any point which is strictly in the interior of the polytope $P(\beta)$ the above function is finite. The path-following algorithm proceeds in several phases. Initially we start with a strictly interior point $x^{(0)}$ which is a good approximation to the analytic center of $P(-\infty)$ and $\beta^{(0)} = -\infty$. The path-following algorithm generates a sequence of parameters $\beta^{(0)}, \beta^{(1)}, \ldots, \beta^{(k-1)}, \beta^{(k)}, \ldots$ whose limit is the maximum value of $c^T x$ over the network polytope P, and also a sequence of points $x^{(0)}, x^{(1)}, \ldots, x^{(k-1)}, x^{(k)}, \ldots$ such that $x^{(k)}$ is a good approximation to the analytic center of $P(\beta^{(k)})$. The final β is very close to the optimal value of $c^T x$ which enables us to isolate the optimal facet of the network polytope together with an exact optimum. As β is varied from $-\infty$ to its maximum possible value in a continuous manner, a continuous trajectory is generated that would start from the analytic center of the network polytope P and terminate at a point that maximizes the objective function $c^T x$ over P. This trajectory is characterized by

$$\Delta\tau + tc = A^T \lambda$$

where $t \in \Re$, $t \geq 0$. $\Delta \in \Re^{e \times e}$ is a diagonal matrix whose ith diagonal entry equals $1/(l_i - x_i) + 1/(u_i - x_i)$ (See [46, 45, 38] for more details). $\tau \in \Re^e$ is a vector of all 1's and $\lambda \in \Re^{v-1}$ is a vector of node potentials. Note that here $\Delta\tau$ can be interpreted as a vector of "reduced costs". In each

phase, we find $x^{(k)}$ from $x^{(k-1)}$ by applying Newton's method to the logarithmic barrier function $F(x, \beta^{(k)})$ starting from $x^{(k-1)}$.

A Step of Newton's Method A step of Newton's method applied to minimizing $F(x, \beta)$ proceeds as follows. Let $\tilde{x}$ be the current point. Let η be the gradient and Q be the Hessian of $F(x, \beta)$ at the current point . We first compute a direction ξ by solving the local optimization problem

$$\max \ -\eta^T \xi$$

subject to

$$A\xi = 0$$

and

$$\xi^T Q \xi \leq 1 \ ,$$

and then (approximately) minimize $F(x, \beta)$ on the line $\tilde{x} + t\xi$. For details see [45]. During the line minimization, we express the logarithmic barrier function as a function of a single parameter t and then do a bisection search.

Isolating an Exact Optimum Since the capacities are integral, the analytic center has the property that any edge flow that is not fixed to either the upper or the lower bound on the optimal facet must be at least $1/(e+1)$ away from both the bounds (See [46, 45, 38] for details). Using this property we adaptively try to isolate an optimal point in the following manner. Suppose that we have a good approximation to the center of the polytope $P(\beta)$ for some β (for a measure of goodness of the approximation see section 2.4). Then any edge flow that is less than $1/(2e)$ from either the upper (resp. lower) bound is guaranteed to be fixed to the upper (resp. lower) bound at the optimum (See [46, 45, 38] for details). We fix each such edge flow to the closer of the two bounds and obtain a flow that may not satisfy the conservation constraints $Ax = b$.

Let

$$x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \quad l = \begin{bmatrix} l_1 \\ l_2 \end{bmatrix} \quad u = \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} \quad A = \begin{bmatrix} A_1 & A_2 \end{bmatrix}$$

where subscript 1 indicates those edges fixed to appropriate bounds. Let $\tilde{x} = \begin{bmatrix} \tilde{x}_1 \\ \tilde{x}_2 \end{bmatrix}$ denote the solution before any edge flows are fixed to upper or lower bounds. Let $\hat{x}_1$ be the flow vector obtained from $\tilde{x}_1$ after fixing edge flows to the closer of the two bounds. Since the flow vector $\begin{bmatrix} \hat{x}_1 \\ \tilde{x}_2 \end{bmatrix}$ may not satisfy conservation constraints we need to obtain an $\hat{x}_2$ satisfying the conservation constraints

$$A_2\hat{x}_2 = b - A_1\hat{x}_1 = \hat{b}_2.$$

We obtain $\hat{x}_2$ from $\tilde{x}_2$ by applying a correction to edge flows in a spanning forest (Though any spanning forest will do, we find a maximum spanning forest with edge weights as defined in section 2.3, subsection **Reducing to a Full-Dimensional Form**, since these edges have large slacks). Next we check if $\hat{x} = \begin{bmatrix} \hat{x}_1 \\ \hat{x}_2 \end{bmatrix}$ satisfies the capacity and lower bound constraints. If it does then we have a feasible solution $\hat{x}_2$ to the reduced problem

$$\min \quad c_2^T x_2$$

subject to

$$A_2 x_2 = \hat{b}_2$$

$$l_2 \leq x_2 \leq u_2.$$

The correction to $\tilde{x}_2$ may be computed as follows. Let

$$x_2 = \begin{bmatrix} x_{2_f} \\ x_{2_o} \end{bmatrix}$$

where x_{2_f} corresponds to edges in a spanning forest and x_{2_o} corresponds to edges outside the forest (Here the spanning forest is in the network defined by the edges corresponding to x_2). We want to find a correction $\varepsilon = \begin{bmatrix} \varepsilon_f \\ 0 \end{bmatrix}$ such that

$$\hat{x}_2 = \tilde{x}_2 + \varepsilon$$

satisfies conservation constraints. Let

$$c_2 = \begin{bmatrix} c_{2_f} \\ c_{2_o} \end{bmatrix} \quad l_2 = \begin{bmatrix} l_{2_f} \\ l_{2_o} \end{bmatrix} \quad u_2 = \begin{bmatrix} u_{2_f} \\ u_{2_o} \end{bmatrix} \quad A_2 = \begin{bmatrix} A_{2_f} & A_{2_o} \end{bmatrix}.$$

The correction ε may be obtained by solving the linear system

$$A_{2_f}\varepsilon_f = \hat{b}_2 - A_2\tilde{x}_2.$$

This may be done in linear time since A_{2_f} corresponds to a spanning forest.

By further eliminating the variables x_{2_f} (i.e. those in a spanning forest) we can reduce the problem to full-dimensional form:

$$\min \quad (c_{2_o} - A_{2_o}^T(A_{2_f}^{-1})^T c_{2_f})^T x_{2_o}$$

subject to

$$l_{2_f} \leq A_{2_f}^{-1}(\hat{b}_2 - A_{2_o}x_{2_o}) \leq u_{2_f}$$

$$l_{2_o} \leq x_{2_o} \leq u_{2_o}.$$

If the reduced cost $c_{2_o} - A_{2_o}^T(A_{2_f}^{-1})^T c_{2_f}$ is zero (we test for numerical zero) then the flow $\hat{x}$ is guaranteed to be optimal.

To summarize, we carry out the following procedure after every centering step in the final phases of the algorithm. Fix all the flows that are within $1/(2e)$ from the upper (resp. lower) bound to the upper (resp. lower) bound. Find a maximum spanning forest in the network with the edges corresponding to these fixed variables removed. Apply a correction to the variables in this spanning forest (as described above) and check if the resulting flow is feasible. If this flow is feasible, we have a facet, namely, the one where the variables in x_1 are fixed, which is a candidate for being the optimal facet. If the reduced cost function $c_{2_o} - A_{2_o}^T(A_{2_f}^{-1})^T c_{2_f}$ is zero then this facet is declared to be the optimal and the algorithm stops. Otherwise the algorithm continues. Note that the algorithm always halts with an exact optimum. Specifically, this procedure allows us to isolate the entire optimal facet along with a good approximation to the center of the optimal facet.

If an optimal vertex is desired, the cost vector may be modified by adding a "small" random perturbation. An optimal solution to this modified problem is also optimal for the original problem, and with high probability the optimal facet of the modified problem is a single vertex (See [10]). As this procedure requires too many bits of precision in the perturbed cost vector, it was not implemented. Instead our algorithm isolates a good approximation to the analytic center of the optimal facet.

Getting an Initial Feasible Point To get an initial interior feasible point, the flow in each edge is set midway between the upper and lower bound. Since such a flow may violate the conservation constraints, we introduce extra edges to a distinguished ground node to carry the excess or deficiency from those nodes where the conservation constraint is not satisfied. We put a high cost on each of these edges to ensure that at an optimum the flows in these edges are zero provided the original problem is feasible. We use a couple of simple heuristics to estimate the cost on these newly introduced edges; for example, letting the cost of a new edge be greater than the cost of a Maximum Spanning Tree (MST) of (absolute) costs suffices.

2.3 Solving the Linear System at Each Step

During each step of the Newton's Method (while computing the center), we have to find a direction ξ which is the solution to the local optimization problem

$$\max \ -\eta^T \xi$$

subject to

$$A\xi = 0$$

and

$$\xi^T Q \xi \leq 1 \ ,$$

Matrix Q is the Hessian of the logarithmic barrier function $F(x, \beta)$ and can be expressed as $D + \hat{w} c c^T$ where D is a diagonal matrix whose ith diagonal entry is

$$D_{ii} = \frac{1}{(x_i - l_i)^2} + \frac{1}{(u_i - x_i)^2}$$

and $\hat{w} = w/(c^T x - \beta)^2$ where w is a problem dependent multiplier. The above problem can be reduced to full-dimensional form by eliminating the flow variables in a spanning tree using the conservation constraints $A\xi = 0$.

Reducing to a Full-Dimensional Form Let $\tilde{x}$ be the current point and $\tilde{D}$ the matrix D evaluated at $\tilde{x}$. Let the weight of edge i with respect to the current point $\tilde{x}$ be defined as $D_{ii}^{-\frac{1}{2}}$. We reduce the above problem to full-dimensional form by eliminating the flow variables corresponding to the edges in a specific spanning tree, namely a Maximum Spanning Tree(MST) with edge weights defined above. Intuitively the MST edges are those edges that have the "largest" slacks for changing the flow. We split the variables of ξ into two blocks - those belonging to the MST, denoted by ξ_z, and those that do not, denoted by ξ_y. Let

$$\begin{aligned}
\xi &= \begin{bmatrix} \xi_z \\ \xi_y \end{bmatrix} \\
A &= \begin{bmatrix} A_z & A_y \end{bmatrix} \\
D &= \begin{bmatrix} D_z & 0 \\ 0 & D_y \end{bmatrix} \\
c &= \begin{bmatrix} c_z \\ c_y \end{bmatrix}
\end{aligned}$$

be the block representation of the vectors and matrices involved in the above local optimization problem, where subscript z (y) represents the columns/rows corresponding to the MST (non-MST) variables. Let $B = -A_z^{-1} A_y$. Eliminating the MST variables reduces the above local optimization problem to

$$\max \; -\hat{\eta}^T \xi_y$$

subject to

$$\xi_y^T \hat{Q} \xi_y \leq 1 \; , \tag{2.1}$$

where $\hat{\eta} = B^T \eta_z + \eta_y$, $\hat{Q} = B^T D_z B + D_y + \hat{w}\hat{c}\hat{c}^T$, and $\hat{c} = B^T c_z + c_y$.

The direction ξ_y may be obtained by solving the linear system

$$\hat{Q}\xi_y = -\hat{\eta} \tag{2.2}$$

and then ξ_z may be computed as $\xi_z = B\xi_y$. We solve the linear system (2.2) by a Pre-conditioned Conjugate Gradient (PCCG) method (See [20]) with $M = D_y + \hat{w}\hat{c}\hat{c}^T$ as the pre-conditioner. Note that $\hat{Q}$ may be expressed as

$$\hat{Q} = M + B^T D_z B$$

Since we eliminated variables ξ_z corresponding to an MST, it is guaranteed that the absolute value of any entry in $D_z^{\frac{1}{2}} B D_y^{-\frac{1}{2}}$ is bounded by 1 and thereby the trace of $M^{-\frac{1}{2}} B^T D_z B M^{-\frac{1}{2}}$ is bounded by ve. The bound on any entry in $D_z^{\frac{1}{2}} B D_y^{-\frac{1}{2}} = D_z^{\frac{1}{2}} A_z^{-1} A_y D_y^{-\frac{1}{2}}$ may be obtained as follows:

Consider a column a_y of A_y corresponding to a non-tree edge e. Let Π be the unique path in the MST between the end points of e. Note that a numbering of the columns of A_z naturally induces a numbering of the edges in the MST. The ith entry in $A_z^{-1} a_y$ is ± 1 if the ith edge in the MST lies on the path Π; otherwise the entry is zero. So a non-zero entry in $D_z^{\frac{1}{2}} A_z^{-1} a_y D_y^{-\frac{1}{2}}$ is of the form w_e/w where w_e is the weight of edge e and w is the weight of some edge on path Π. It then follows that the absolute value of any entry in $D_z^{\frac{1}{2}} A_z^{-1} A_y D_y^{-\frac{1}{2}}$ is bounded by 1 (See [47] for details).

The above argument ensures that M is a good pre-conditioner for $\hat{Q}$. As we approach a vertex solution, entries in $D_z^{\frac{1}{2}}$ remain bounded because they correspond to edges with non-zero slack whereas entries in $D_y^{-\frac{1}{2}}$ approach zero. Hence, the closer we are to a vertex solution, the better is the pre-conditioner.

In our experiments, it has been observed that the PCCG with this pre-conditioner M halts in about 60-70 iterations on the average (with a solution of desired accuracy) for problems with up to 250,000 edges. Note that solving a linear system $Mu = r$ can be trivially accomplished in $O(e)$ operations, and we never need to form the matrix $\hat{Q}$ (or any other matrix) explicitly. Moreover, multiplication of $\hat{Q}$, A or A^T by a vector may be performed in $O(e)$ operations, and $A_z u = r$ may be solved in $O(v)$ operations, using a standard graph adjacency list data structure.

MST Computation For the sake of efficiency, we round off the edge weights to multiples of $(1+\delta)$ for a suitable δ and use Bucket Sorting coupled with Kruskal's MST algorithm (See [12]). Using such an approximate MST - instead of an exact MST - only slightly degrades the quality of the pre-conditioner M.

Reducing the Constants in PCCG All the intermediate vectors generated in our implementation of PCCG lie in a Krylov Space of dimension at most $v+1$ rather than e. This may be seen as follows:

Note that $\hat{Q}$ may be written as $\hat{Q} = D_y + UU^T$ where $U = [\hat{w}^{\frac{1}{2}}c \quad B^T D_z^{\frac{1}{2}}]$, and $U \in \Re^{e \times v}$. Let $\tilde{Q} = D_y^{-\frac{1}{2}} \hat{Q} D_y^{-\frac{1}{2}} = I + \tilde{U}\tilde{U}^T$ where $\tilde{U} = D_y^{-\frac{1}{2}} U$. Instead of directly solving $\hat{Q}\xi_y = -\hat{\eta}$ we first compute $\tilde{\xi}$ by solving

$$\tilde{Q}\tilde{\xi} = -D_y^{-\frac{1}{2}}\hat{\eta} = \tilde{\eta}$$

and then compute $\hat{\xi}$ as $\hat{\xi} = D_y^{\frac{1}{2}}\tilde{\xi}$. During the solution of $\tilde{Q}\tilde{\xi} = \tilde{\eta}$ by PCCG successive vectors are generated as linear combinations of previous vectors or by multiplying previous vectors by M^{-1} or $\tilde{Q}$ (See [20]). The initial vector is $\tilde{\eta}$. Note that if $x \in span\{\tilde{\eta}, \tilde{u}_1, \ldots, \tilde{u}_v\}$ where $\tilde{u}_1, \ldots, \tilde{u}_v$ are the columns of $\tilde{U}$ then both $M^{-1}x$ and $\tilde{Q}x$ lie in the span of $\{\tilde{\eta}, \tilde{u}_1, \ldots, \tilde{u}_v\}$. As a result all the vectors generated during PCCG lie in the Krylov space given by $span\{\tilde{\eta}, \tilde{u}_1, \ldots, \tilde{u}_v\}$ and may be implicitly represented by $v+1$ coefficients.

Thus most of the intermediate calculations may be implicitly performed using vectors of size $v+1$; this leads to reduced constants in the running time as well as storage and to an improvement in practical performance.

2.4 Some Techniques to Improve Speed and Stability

In our implementation we use several techniques to gain additional speed and improve the stability of the algorithm. We describe them in this section.

Approximate Centering In the initial phases of the algorithm, it suffices to compute a very coarse analytic center. A good approximation to the center is required only when the algorithm

is getting ready to isolate the optimal facet. So in the initial phases we find a very coarse approximation to the center which is sufficiently far from the objective function hyperplane, and eventually switch to an almost exact centering when the width of the polytope in the direction of the cost vector falls below a certain threshold. Closeness to the center is measured by the quantity $\hat{\eta}^T\hat{Q}^{-1}\hat{\eta}$. Initially a coarse approximation x to the center such that $\hat{\eta}^T\hat{Q}^{-1}\hat{\eta} \leq \epsilon e$ (ϵ a small constant) suffices. Later on when we are close to isolating the optimal facet, we switch to a finer approximation x satisfying the condition $\hat{\eta}^T\hat{Q}^{-1}\hat{\eta} \leq 0.5$. Note that for a point x close to the center ω, $\hat{\eta}^T\hat{Q}^{-1}\hat{\eta} \approx 2(F(x,\beta) - F(\omega,\beta))$ (See [45, 38, 32] for theoretical results). The empirical data suggests that in practice the complexity of the two centerings is much better than the theoretical bounds.

Variable PCCG Stopping Threshold The farther we are from the analytic center the larger may be the error in the Newton direction ξ without degrading the rate of convergence to the center. As a result, stopping (error) threshold for the PCCG can be coarser in the beginning when we are far from the center and may be progressively tightened as we approach the center. The PCCG as implemented in our algorithm stops when

$$r^T M^{-1} r \leq \mathit{threshold}$$

where r is the current residue and M is the preconditioner as defined earlier. When we are not too close to the center i.e. $\hat{\eta}^T\hat{Q}^{-1}\hat{\eta} \geq 0.1w$ where w is the weight on the objective function plane, the *threshold* is set to $\max\{10^{-4}w, 0.1\}$. When we are close to the center i.e. $\hat{\eta}^T\hat{Q}^{-1}\hat{\eta} < 0.1w$, the *threshold* is set to 0.1.

Reducing the Problem Size Since the capacities are integral, the center has the property that any edge flow that is not fixed to either the upper or the lower bound on the optimal facet must be at least $1/(e+1)$ away from both the bounds. This may be argued as follows:

At the analytic center ω of the polytope $P(\beta) = P \cap \{x : c^T x \geq \beta\}$ we have that

$$\sum_{i=1}^{e} \left(\frac{u_i - x_i}{u_i - \omega_i} + \frac{x_i - l_i}{\omega_i - l_i} \right) + w\frac{c^T x - \beta}{c^T\omega - \beta} = w + 2e$$

(See [46, 38] for details). At any optimum point $x^\star$, we have that $c^T x^\star - \beta \geq c^T \omega - \beta$. Thus

$$\sum_{i=1}^{e} \left(\frac{u_i - x_i^\star}{u_i - \omega_i} + \frac{x_i^\star - l_i}{\omega_i - l_i} \right) \leq 2e$$

Also, each term in the summation on the left hand side in the above inequality is at least 1. Thus for each i and each optimal point $x^\star$ we have

$$\frac{u_i - x_i^\star}{u_i - \omega_i} + \frac{x_i^\star - l_i}{\omega_i - l_i} \leq e + 1 \tag{2.3}$$

Furthermore, for any coordinate x_i either x_i is fixed to l_i or u_i on the optimal facet or the width of the optimal facet in the direction of x_i is at least 1 since vertices have integer coordinates. Suppose x_i is not fixed on the optimal facet. Then there exist optimal points x' and x'' such that $u_i - x_i' \geq 1$ and $x_i'' - l_i \geq 1$. Then from inequality (2.3) it follows that $u_i - \omega_i \geq 1/(e+1)$ and $\omega_i - l_i \geq 1/(e+1)$.

Using this property, we fix those edge flows which at the current approximate center are closer than $1/(2e)$ to a bound to that bound. For these edges the optimal flow value is the bound it is fixed to. The edges that are fixed may be removed from further consideration in the algorithm.

Dynamic Shift of Origin for Stability If either the upper or the lower bound of an edge is large in magnitude there is a chance of numerical instability as the edge flow approaches a bound in the final stages of the algorithm. A typical machine has about 15 to 16 decimal digits of precision. However, if we have a capacity of 10^8 we effectively could loose half the digits of precision. For example, suppose the current flow in an edge is 10^{-8} less than the capacity which is 10^8 and the machine has 16 decimal digits of precision. Now if the flow moves further towards the capacity, the flow value would be set to the capacity as a result of numerical error thereby leading to a division by zero in the intermediate calculations since the algorithm assumes a strictly interior point. To remove instabilities of this kind, we apply a dynamic shift-of-origin changing those bounds on edges which have a flow close to them to zero and working with the offset of the flow on such edges. At the end of the algorithm, the origin is shifted back to get the correct flows. As a result, an edge flow can approach a bound to essentially within machine precision. Note that even though the analytic centers generated during the algorithm cannot be too close to a boundary, the intermediate points

generated in moving from one center to the next get quite close to some of the boundaries. It is because of this that dynamic shift of origin is required for large problems.

2.5 Experimental Results and Analysis

MINFLOW, our implementation for minimum cost flow, was tested on several problems using a Sun Sparc-2 workstation running SunOS 4.1.1 with FPU version 3. It was experimentally observed that vector inner product and saxpy operations ran at about 0.7 to 1.0 MEGAFLOPS. In order to estimate the speed of pointer manipulations several examples of standard breadth-first-search using adjacency list representation were run, and the time for breadth-first-search was estimated to be about 3 microseconds per edge. All maximum flow problems were converted to minimum cost flow problems and then solved using MINFLOW. The following tables and graphs show the experimental data along with timing and iteration counts. I/O times are not included in the timing data. The time and iteration counts to find the optimal solution are averaged over the number of instances of each size that was run.

Seeds were selected according to the following rule for all families reported except mesh-standard-one, mesh-standard-two, mesh-standard-four and mesh-standard-eight. Suppose the total number of instances for a given size is I. Then the seed for instance i, $1 \leq i \leq I$, is given by the formula $5 + (i-1) * 2$. For families mesh-standard-one, mesh-standard-two, mesh-standard-four and mesh-standard-eight the formula is $1 + (i-1) * 2$.

One of the referees pointed out that NETFLO uses a slow insertion sort while reading in the data and suggested that we run bench marks to see how this affects the comparisons between MINFLOW and NETFLO. From each family we selected two instances of moderate size and timed the initial insertion sort in the NETFLO code. It was found that for families capt-hi-bits, capt-hi-bits-mod, capt-lo-bits and capt-lo-bits-mod the percentage time spent in the initial insertion sort is in the range 60%–64%. For washing-line-moderate this percentage is 47%. For acyclic-dense, genrmf-long and grid-long this initial time is moderately small (in the range of 13%–25%). For the remaining eleven families, this time is negligible (below 8%). This fact should be taken into consideration while examining the timing data for NETFLO.

The data for each input class consists of:

- A table which gives the absolute timing statistics for MINFLOW, NETFLO and RELAXT III along with the standard deviation.

- A table which gives the iteration count statistics for MINFLOW, NETFLO and RELAXT III along with the standard deviation. Each iteration for MINFLOW is a PCCG iteration in which the work done is proportional to the number of edges. A NETFLO iteration consists of moving from one vertex to to another in the network simplex algorithm [33]. A RELAXT III iteration corresponds to a relaxation step as described in [3, 6, 4, 5].

- Graphs for comparison with NETFLO and RELAXT III codes. The average of the running time ratios are plotted against problem size represented by the number of nodes on a log-log scale. This is done because for small problems, both NETFLO and RELAXT III are typically much faster than MINFLOW but as the problem size increases the situation reverses for several input classes. For input sizes where the graph is above the abscissa of 1.0, MINFLOW is slower compared to the other code (NETFLO or RELAXT III) and in the region where the graph is less than the abscissa of 1.0, MINFLOW is faster. Note that we were unable to obtain some of the data points for comparison with RELAXT III because the RELAXT III code failed to solve those problems.

Note that we have shortened RELAXT III to RELAXT in the tables and graphs.

One of the referees also suggested modifying the parameters to the CAPT generator so that the number of sinks is roughly $N/2$ where N is the number of vertices minus two. So the CAPT families were run with two types of parameter settings: one, as per DIMACS recommendations, the results of which are given under capt-lo-bits and capt-hi-bits; and two, as per the referee's recommendations, the results of which are summarized under capt-lo-bits-mod and capt-hi-bits-mod. In particular, the parameters *flow* and *intlen* were chosen as *flow*$= 2*$*intlen*$= 2000$ for capt-lo-bits-mod and capt-hi-bits-mod.

We have included the results for the maximum flow problem families for comparison purposes only. So, while examining the statistics for families acyclic-dense, washing-line-moderate, genrmf-long and genrmf-wide, which are maximum flow problems converted to mincost flow problems, the fact that *the simplex algorithm is known to perform poorly on mincost flow problems derived from*

maximum flow problems should be borne in mind. Furthermore, inclusion of these results indicate that Interior Point methods work well uniformly irrespective of the origins of the problem.

The smaller problems give trends for the comparative growth rates for the three codes. Since it was too time consuming to run many instances for the large problem sizes, we typically ran 1 or 2 instances only. The main memory in our machine was sufficient to run problems with a little over a 100,000 arcs. For all input classes we ran as big an instance as would fit in virtual memory without thrashing. As a result only a few data points were obtained for input classes acyclic-dense, capt-lo-bits, capt-lo-bits-mod, capt-hi-bits and capt-hi-bits-mod for which the number of arcs grew quadratically with the number of nodes.

A cursory analysis indicates a growth rate of about $O(e^{1.5})$ for MINFLOW on all the problems.

For input families washing-line-moderate, genrmf-long, genrmf-wide, grid-den-sixteen, grid-inc-den, grid-square, and grid-wide, the graphs indicate that MINFLOW has a slower growth rate than both NETFLO and RELAXT III with increasing problem size.

For grid-den-eight, MINFLOW is faster than NETFLO and RELAXT III, even for small problem sizes.

For acyclic-dense and grid-long, MINFLOW has faster growth rate than both NETFLO and RELAXT III.

For capt-lo-bits, capt-lo-bits-mod, capt-hi-bits and capt-hi-bits-mod only four data points for each family have been obtained due to memory restrictions. The trend seems to be that MINFLOW has lower growth rate than NETFLO but higher than RELAXT III for these families.

For mesh-standard-one, mesh-standard-two, mesh-standard-four and mesh-standard-eight MIN-FLOW has lower growth rate than NETFLO but seems to have the same growth rate as RELAXT III.

2.6 Conclusions

The implementation is quite robust, numerically stable, and fairly insensitive to variations in costs and capacities. Furthermore, we can handle problems with non-integer (real) costs since the stopping criterion does not depend on integrality of costs.

Size			Avg. Time (seconds)			Std. Dev. Time (seconds)		
Nodes	Arcs	#	MINFLOW	NETFLO	RELAXT	MINFLOW	NETFLO	RELAXT
64	2016	13	19.2	0.8	0.7	1.0	0.0	0.1
128	8128	13	141.0	3.7	2.8	9.5	0.1	0.7
256	32640	13	1049.1	21.0	21.3	28.3	0.7	14.7

Size			Iterations			Iterations (Std.dev)		
Nodes	Arcs	#	MINFLOW	NETFLO	RELAXT	MINFLOW	NETFLO	RELAXT
64	2016	13	349.9	794.7	103.5	15.8	93.0	46.4
128	8128	13	812.8	2761.5	177.8	63.8	353.9	88.0
256	32640	13	2002.8	4861.9	457.5	53.7	669.5	226.6

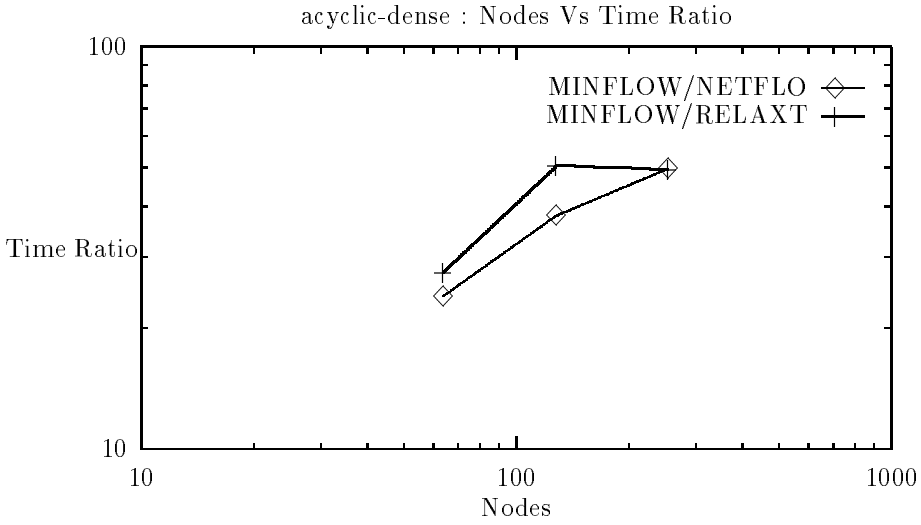


Figure 2.1: Statistics for family **acyclic-dense**

Size			Avg. Time (seconds)			Std. Dev. Time (seconds)		
Nodes	Arcs	#	MINFLOW	NETFLO	RELAXT	MINFLOW	NETFLO	RELAXT
66	127	10	1.7	0.1	0.0	0.2	0.0	0.0
130	373	10	5.3	0.3	0.2	0.4	0.0	0.0
258	1001	10	16.4	1.0	0.9	1.3	0.1	0.2
514	3006	10	67.1	5.4	4.5	15.4	0.3	0.7
1026	8071	10	226.8	27.7	23.6	37.3	1.2	3.3
2050	24276	10	708.8	193.7	136.2	90.5	6.4	20.0
4098	65002	1	3464.6	1088.6	621.7	0.0	0.0	0.0

Size			Iterations			Iterations (Std.dev)		
Nodes	Arcs	#	MINFLOW	NETFLO	RELAXT	MINFLOW	NETFLO	RELAXT
66	127	10	124.6	86.4	25.2	19.2	9.0	7.7
130	373	10	209.4	261.4	76.7	24.4	37.3	24.0
258	1001	10	338.5	580.6	168.9	29.0	108.1	54.6
514	3006	10	627.8	1282.2	357.5	145.4	134.4	69.4
1026	8071	10	1034.3	2696.7	747.1	178.9	165.4	113.3
2050	24276	10	1412.6	5712.0	1538.2	181.9	476.3	232.2
4098	65002	1	3119.0	11063.0	2820.0	0.0	0.0	0.0

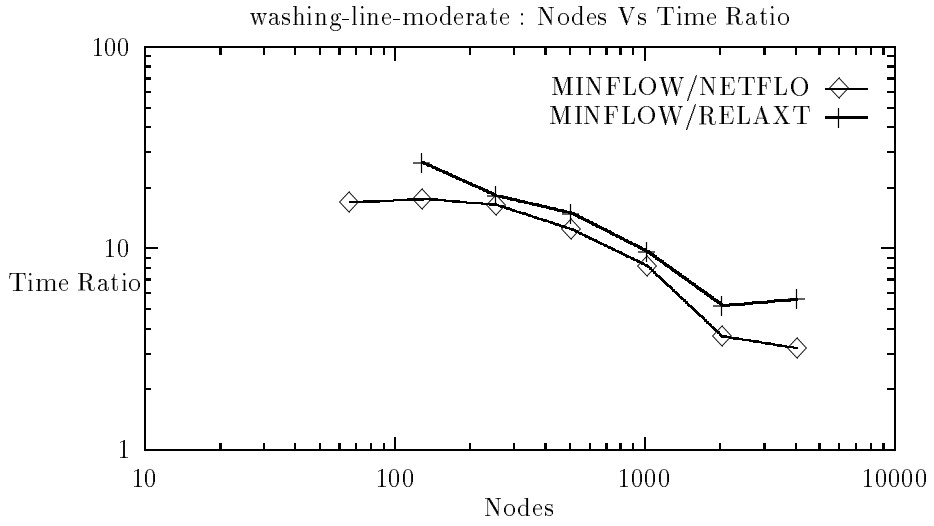


Figure 2.2: Statistics for family **washing-line-moderate**

Size			Avg. Time (seconds)			Std. Dev. Time (seconds)		
Nodes	Arcs	#	MINFLOW	NETFLO	RELAXT	MINFLOW	NETFLO	RELAXT
72	255	10	1.8	0.1	0.1	0.1	0.0	0.0
192	752	10	7.0	0.4	0.5	0.6	0.0	0.0
256	1008	10	10.5	0.6	0.9	0.7	0.0	0.0
575	2390	10	30.7	3.0	4.5	0.7	0.1	0.5
1152	4956	10	80.5	10.8	19.8	3.9	0.3	1.0
2254	9933	10	182.2	41.2	80.7	6.8	1.5	3.6
4096	18368	10	406.7	120.9	276.0	17.0	5.5	24.7
9100	41760	10	1260.0	628.8	1500.2	59.3	33.3	123.6
18432	85872	10	6858.3	2539.0	6958.9	1427.3	65.5	293.3

Size			Iterations			Iterations (Std.dev)		
Nodes	Arcs	#	MINFLOW	NETFLO	RELAXT	MINFLOW	NETFLO	RELAXT
72	255	10	83.6	126.5	41.9	5.5	6.2	7.1
192	752	10	119.0	387.3	123.1	8.2	21.3	13.9
256	1008	10	124.0	496.5	149.8	7.8	21.1	14.3
575	2390	10	165.3	1259.4	394.2	7.8	42.6	41.7
1152	4956	10	225.1	2684.5	857.4	20.9	83.6	54.1
2254	9933	10	250.9	5397.5	1871.4	15.6	159.0	85.8
4096	18368	10	298.5	9641.5	3374.9	20.8	298.7	301.0
9100	41760	10	392.0	22591.8	7763.6	30.8	846.4	590.1
18432	85872	10	730.4	48517.3	16825.2	146.9	862.1	682.3

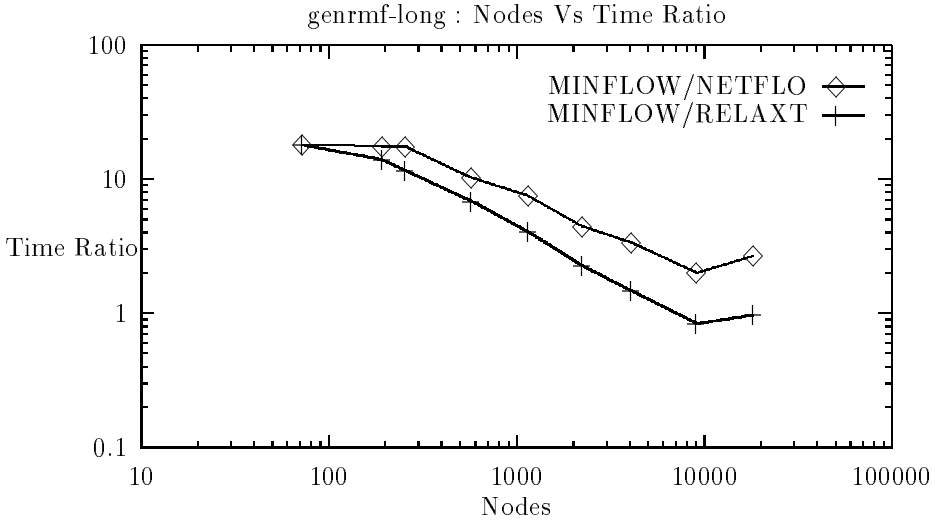


Figure 2.3: Statistics for family **genrmf-long**

Size			Avg. Time (seconds)			Std. Dev. Time (seconds)		
Nodes	Arcs	#	MINFLOW	NETFLO	RELAXT	MINFLOW	NETFLO	RELAXT
72	276	10	2.1	0.1	0.1	0.2	0.0	0.0
147	602	10	5.9	0.3	0.4	0.5	0.0	0.0
300	1280	10	17.0	0.8	1.2	0.9	0.0	0.1
676	3003	10	54.5	3.3	7.0	1.0	0.1	0.5
1024	4608	10	96.5	5.3	16.6	2.1	0.2	1.0
1936	8844	10	243.2	15.4	58.5	12.7	0.5	0.0
3920	18256	10	623.1	59.7		16.2	2.4	
16807	80262	1	10468.2	1308.9		0.0	0.0	

Size			Iterations			Iterations (Std.dev)		
Nodes	Arcs	#	MINFLOW	NETFLO	RELAXT	MINFLOW	NETFLO	RELAXT
72	276	10	124.2	120.4	38.2	9.8	2.2	1.5
147	602	10	179.2	281.0	137.8	12.8	3.7	25.2
300	1280	10	270.6	586.6	330.3	12.9	24.3	22.6
676	3003	10	390.4	1820.8	740.8	8.5	91.9	63.2
1024	4608	10	498.3	2909.8	1142.3	16.5	115.9	56.5
1936	8844	10	708.3	5284.7	2206.0	48.5	208.0	0.0
3920	18256	10	917.2	11785.8		32.9	419.2	
16807	80262	1	3959.0	78634.0		0.0	0.0	

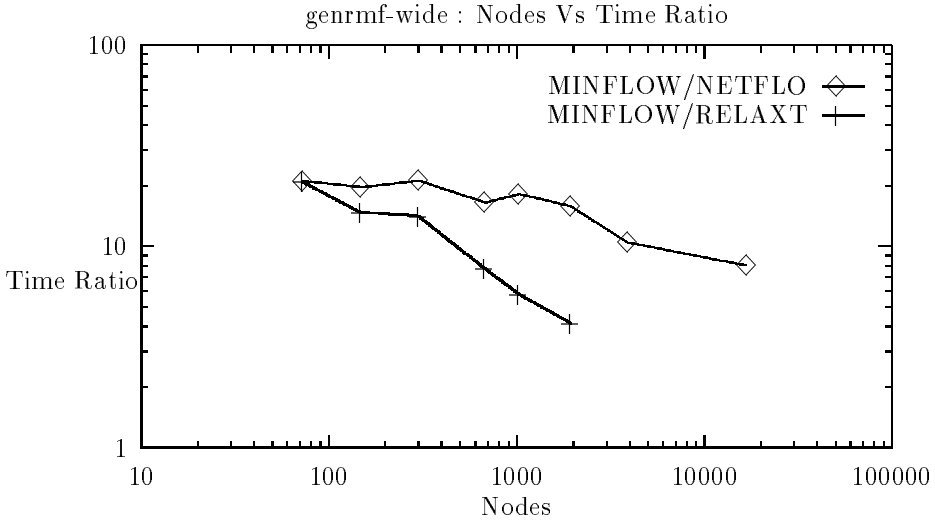


Figure 2.4: Statistics for family **genrmf-wide**

Size			Avg. Time (seconds)			Std. Dev. Time (seconds)		
Nodes	Arcs	#	MINFLOW	NETFLO	RELAXT	MINFLOW	NETFLO	RELAXT
256	2048	10	26.5	2.8	18.2	1.7	0.1	8.8
512	4096	10	56.1	9.9	75.2	1.8	0.4	11.1
1024	8192	10	148.1	34.5	313.4	12.5	0.9	47.1
2048	16384	10	338.5	134.7	1215.2	25.9	7.5	116.2
4096	32768	10	707.2	716.0	4658.0	17.5	27.7	619.1
8192	65536	2	1856.6	2504.3	25876.0	23.7	59.1	1063.1
16384	131072	1	4139.1	10068.9	102480.5	0.0	0.0	0.0
32768	262144	1	16566.1			0.0		

Size			Iterations			Iterations (Std.dev)		
Nodes	Arcs	#	MINFLOW	NETFLO	RELAXT	MINFLOW	NETFLO	RELAXT
256	2048	10	150.2	5878.6	53197.5	20.0	205.7	21464.5
512	4096	10	137.5	15952.7	170049.1	14.2	736.8	26027.1
1024	8192	10	199.8	29290.1	521763.9	18.2	537.4	73833.2
2048	16384	10	235.7	68639.6	1353897.0	40.0	2465.5	131036.8
4096	32768	10	267.4	373951.4	2154553.1	20.5	11898.2	521322.2
8192	65536	2	252.5	331879.0	10645328.0	14.5	2165.0	232605.0
16384	131072	1	354.0	745617.0	29090153.0	0.0	0.0	0.0
32768	262144	1	1576.0			0.0		

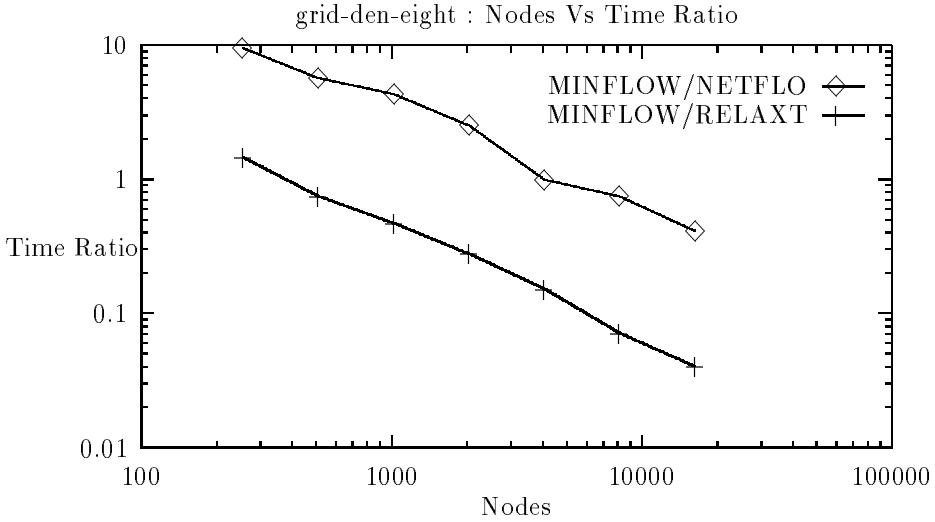


Figure 2.5: Statistics for family **grid-den-eight**

Size			Avg. Time (seconds)			Std. Dev. Time (seconds)		
Nodes	Arcs	#	MINFLOW	NETFLO	RELAXT	MINFLOW	NETFLO	RELAXT
256	4096	8	55.5	5.6	97.7	1.6	0.2	9.5
512	8192	10	132.5	18.7	328.8	7.5	0.3	39.3
1024	16384	10	317.7	74.6	1232.1	16.9	1.6	99.6
2048	32768	10	759.7	287.8	4041.4	20.6	6.0	265.7
4096	65536	10	1666.5	1250.7	15730.9	84.5	38.7	1358.1
8192	131072	2	5517.7	4998.6	58637.1	218.2	63.1	2081.5
16384	262144	1	13211.3	20026.6		0.0	0.0	

Size			Iterations			Iterations (Std.dev)		
Nodes	Arcs	#	MINFLOW	NETFLO	RELAXT	MINFLOW	NETFLO	RELAXT
256	4096	8	202.7	14467.7	165768.2	16.8	701.0	16869.1
512	8192	10	251.9	32430.8	413513.6	17.0	455.5	46136.0
1024	16384	10	347.1	75449.5	1110089.5	16.2	1416.7	129243.0
2048	32768	10	431.7	177102.0	2585272.4	12.8	3244.6	229859.9
4096	65536	10	470.1	459134.6	6166728.4	38.7	13582.9	820072.0
8192	131072	2	697.0	842259.0	14155666.5	4.0	6441.0	577765.5
16384	262144	1	744.0	1949450.0		0.0	0.0	

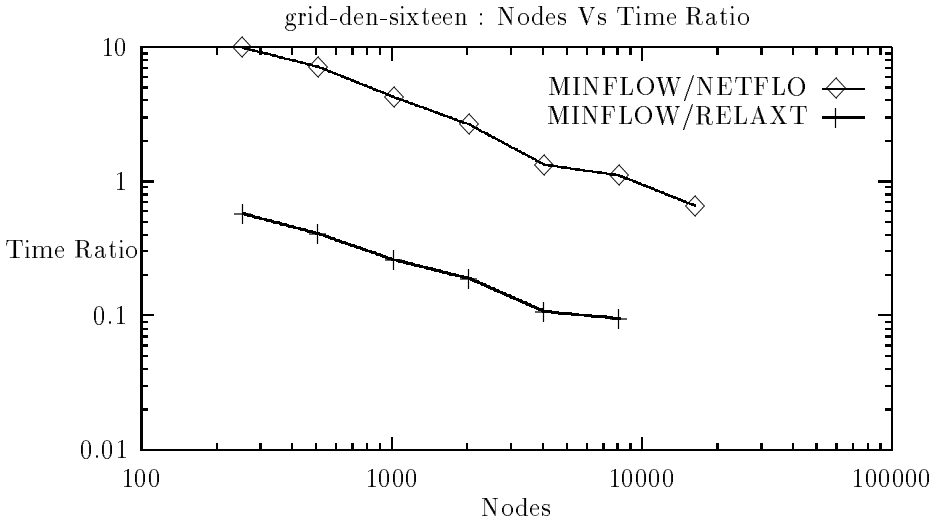


Figure 2.6: Statistics for family **grid-den-sixteen**

Size			Avg. Time (seconds)			Std. Dev. Time (seconds)		
Nodes	Arcs	#	MINFLOW	NETFLO	RELAXT	MINFLOW	NETFLO	RELAXT
256	4096	10	54.9	5.6	97.3	1.8	0.2	9.3
512	11586	10	199.6	29.9	485.1	10.6	0.6	63.6
1024	32768	10	720.0	183.5	2537.8	23.0	3.9	199.1
2048	92682	10	3005.6	1297.4	12802.8	118.2	20.9	457.9
4096	262144	1	12669.0	12088.7	40994.6	0.0	0.0	0.0

Size			Iterations			Iterations (Std.dev)		
Nodes	Arcs	#	MINFLOW	NETFLO	RELAXT	MINFLOW	NETFLO	RELAXT
256	4096	10	201.9	14467.7	165768.2	16.3	701.0	16869.1
512	11586	10	312.4	47781.6	454180.0	20.8	742.0	81517.8
1024	32768	10	527.8	186138.8	1387404.4	21.1	3880.1	165211.7
2048	92682	10	1046.4	637883.4	3666692.0	37.4	34136.0	221686.4
4096	262144	1	1248.0	2514153.0	8293935.0	0.0	0.0	0.0

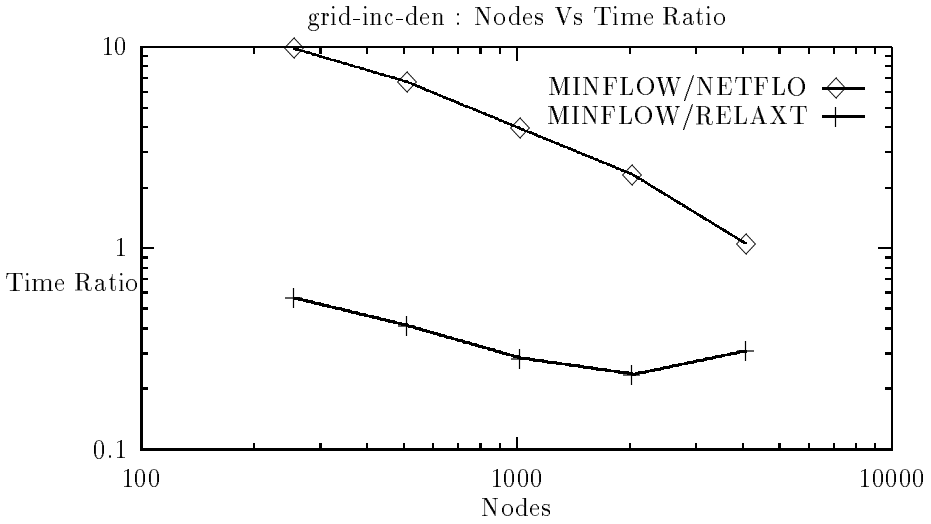


Figure 2.7: Statistics for family **grid-inc-den**

Size			Avg. Time (seconds)			Std. Dev. Time (seconds)		
Nodes	Arcs	#	MINFLOW	NETFLO	RELAXT	MINFLOW	NETFLO	RELAXT
258	512	10	16.5	0.2	0.4	2.4	0.0	0.1
531	1058	10	43.3	0.5	1.3	4.7	0.0	0.1
1026	2048	10	97.5	1.3	4.1	34.3	0.1	0.5
2118	4232	10	280.1	3.4	17.4	20.0	0.3	2.1
4098	8192	10	779.2	9.8	67.8	94.3	0.7	12.6
8283	16562	2	2567.6	36.2	315.5	261.5	4.6	12.0
16386	32768	1	6938.3	118.0	1244.4	0.0	0.0	0.0
33126	66248	1	21711.0	558.9	5608.4	0.0	0.0	0.0

Size			Iterations			Iterations (Std.dev)		
Nodes	Arcs	#	MINFLOW	NETFLO	RELAXT	MINFLOW	NETFLO	RELAXT
258	512	10	309.7	346.5	1357.5	37.0	43.1	334.7
531	1058	10	433.0	826.9	3353.7	47.8	79.2	570.0
1026	2048	10	514.4	1968.3	7620.6	179.0	123.9	1323.2
2118	4232	10	814.9	4916.6	13886.0	46.6	297.1	4367.4
4098	8192	10	1221.3	11509.7	29922.8	134.2	625.8	11600.2
8283	16562	2	2112.5	30226.0	136411.5	208.5	1814.0	3973.5
16386	32768	1	2831.0	70167.0	377766.0	0.0	0.0	0.0
33126	66248	1	4597.0	197148.0	1020525.0	0.0	0.0	0.0

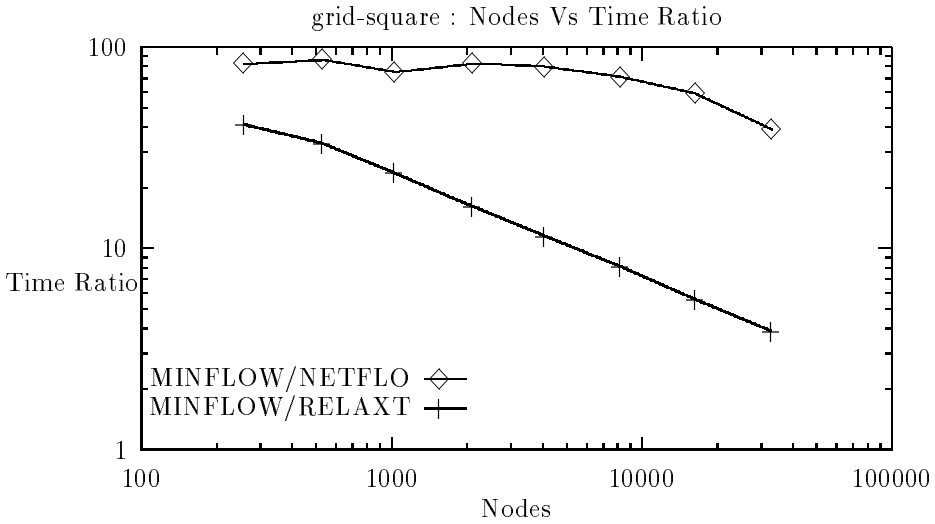


Figure 2.8: Statistics for family **grid-square**

Size			Avg. Time (seconds)			Std. Dev. Time (seconds)		
Nodes	Arcs	#	MINFLOW	NETFLO	RELAXT	MINFLOW	NETFLO	RELAXT
258	512	10	16.6	0.2	0.4	2.3	0.0	0.1
514	1008	10	43.6	0.5	1.0	5.0	0.0	0.2
1026	2000	10	132.3	1.0	2.5	5.8	0.0	0.2
2050	3984	10	364.4	2.2	6.6	43.6	0.1	1.1
4098	7952	10	978.6	5.1	19.7	42.2	0.4	2.9
8194	15888	2	3367.1	10.5	43.4	147.7	0.5	3.3
32770	63504	1	36619.2	57.6	374.4	0.0	0.0	0.0

Size			Iterations			Iterations (Std.dev)		
Nodes	Arcs	#	MINFLOW	NETFLO	RELAXT	MINFLOW	NETFLO	RELAXT
258	512	10	309.7	346.5	1357.5	37.0	43.1	334.7
514	1008	10	463.7	686.4	1726.6	40.0	110.8	936.8
1026	2000	10	776.8	1335.1	790.2	33.6	95.0	238.3
2050	3984	10	1140.2	2353.1	842.3	154.5	137.5	374.0
4098	7952	10	1595.0	5096.2	1500.9	107.1	633.1	878.0
8194	15888	2	2767.5	9056.0	1382.0	147.5	849.0	158.0
32770	63504	1	6500.0	25243.0	6023.0	0.0	0.0	0.0

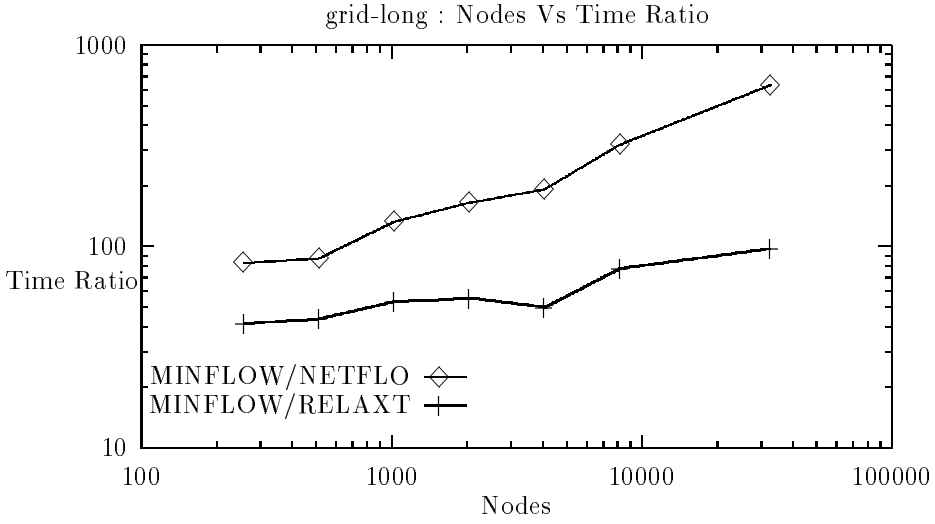


Figure 2.9: Statistics for family **grid-long**

Size			Avg. Time (seconds)			Std. Dev. Time (seconds)		
Nodes	Arcs	#	MINFLOW	NETFLO	RELAXT	MINFLOW	NETFLO	RELAXT
258	512	10	16.4	0.2	0.4	2.3	0.0	0.0
514	1040	10	41.4	0.6	1.5	5.3	0.0	0.1
1026	2096	10	106.1	1.4	6.0	14.6	0.1	0.7
2050	4208	10	222.2	3.9	25.3	25.2	0.3	3.3
4098	8432	10	529.0	12.1	111.7	33.7	0.8	10.8
8194	16880	2	1327.4	41.7	589.3	99.7	0.2	70.2
16386	33776	1	2925.1	166.2	2501.6	0.0	0.0	0.0
32770	67568	1	6617.7	719.1	12477.0	0.0	0.0	0.0
65538	135152	1	20737.5	2578.5	66517.5	0.0	0.0	0.0

Size			Iterations			Iterations (Std.dev)		
Nodes	Arcs	#	MINFLOW	NETFLO	RELAXT	MINFLOW	NETFLO	RELAXT
258	512	10	309.7	346.5	1357.5	37.0	43.1	334.7
514	1040	10	387.1	1090.7	4444.6	49.6	139.7	656.5
1026	2096	10	501.9	2618.6	17748.9	51.3	312.5	3199.2
2050	4208	10	559.9	6132.2	55576.4	52.7	523.0	9823.2
4098	8432	10	688.1	14204.1	156956.9	48.0	770.5	20308.4
8194	16880	2	845.5	33238.5	550904.5	78.5	138.5	85058.5
16386	33776	1	932.0	73542.0	1263055.0	0.0	0.0	0.0
32770	67568	1	1080.0	170984.0	4595085.0	0.0	0.0	0.0
65538	135152	1	1469.0	346385.0	13321493.0	0.0	0.0	0.0

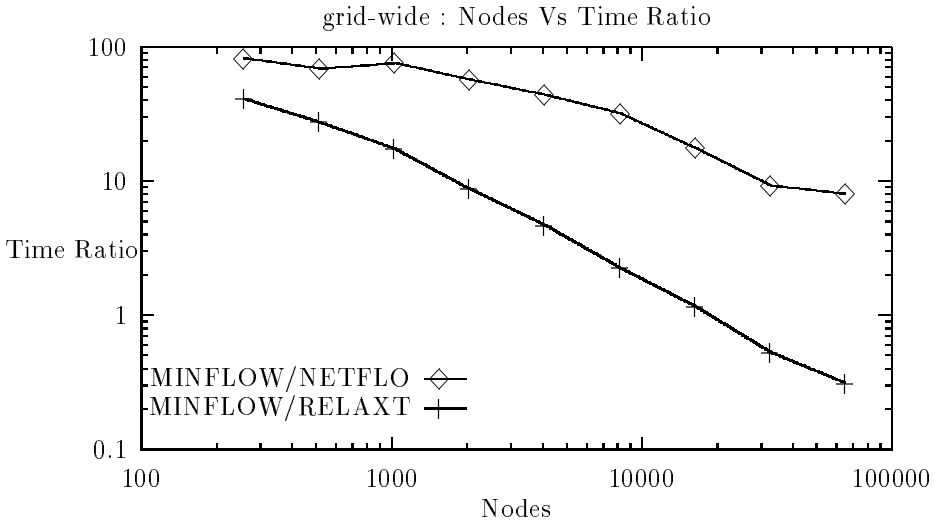


Figure 2.10: Statistics for family **grid-wide**

Size			Avg. Time (seconds)			Std. Dev. Time (seconds)		
Nodes	Arcs	#	MINFLOW	NETFLO	RELAXT	MINFLOW	NETFLO	RELAXT
130	2113	10	16.1	1.5	0.5	2.6	0.0	0.0
258	8321	10	87.4	11.5	2.0	13.3	0.4	0.0
514	33025	10	546.0	93.8	7.9	69.5	2.1	0.1
1026	131585	2	4273.5	812.9	33.8	270.1	43.9	0.4

Size			Iterations			Iterations (Std.dev)		
Nodes	Arcs	#	MINFLOW	NETFLO	RELAXT	MINFLOW	NETFLO	RELAXT
130	2113	10	92.8	1.6	0.3	35.6	0.5	0.5
258	8321	10	175.3	4.2	3.1	44.8	2.6	2.3
514	33025	10	378.5	21.6	20.7	46.7	3.2	3.7
1026	131585	2	1198.0	90.5	83.0	29.0	5.5	7.0

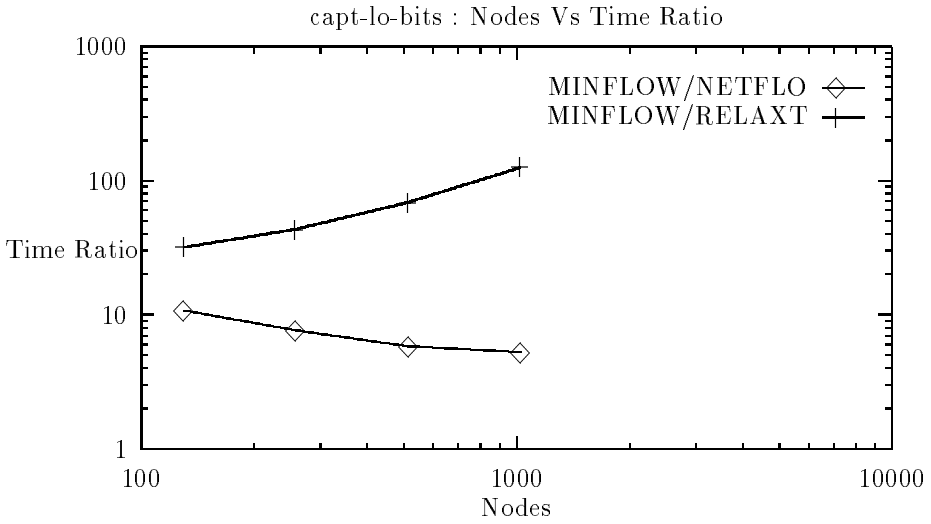


Figure 2.11: Statistics for family **capt-lo-bits**

Size			Avg. Time (seconds)			Std. Dev. Time (seconds)		
Nodes	Arcs	#	MINFLOW	NETFLO	RELAXT	MINFLOW	NETFLO	RELAXT
130	2113	10	24.7	1.6	0.5	2.0	0.1	0.0
258	8321	10	133.0	11.7	2.2	11.3	0.4	0.0
514	33025	10	801.4	94.1	8.7	54.7	2.3	0.1
1026	131585	2	3513.7	820.5	38.5	125.0	5.2	0.1

Size			Iterations			Iterations (Std.dev)		
Nodes	Arcs	#	MINFLOW	NETFLO	RELAXT	MINFLOW	NETFLO	RELAXT
130	2113	10	207.0	335.6	171.3	17.5	34.4	11.9
258	8321	10	312.5	681.7	369.1	22.4	41.0	23.0
514	33025	10	510.2	1316.6	765.1	39.8	45.2	17.1
1026	131585	2	674.5	2702.5	1574.0	29.5	42.5	12.0

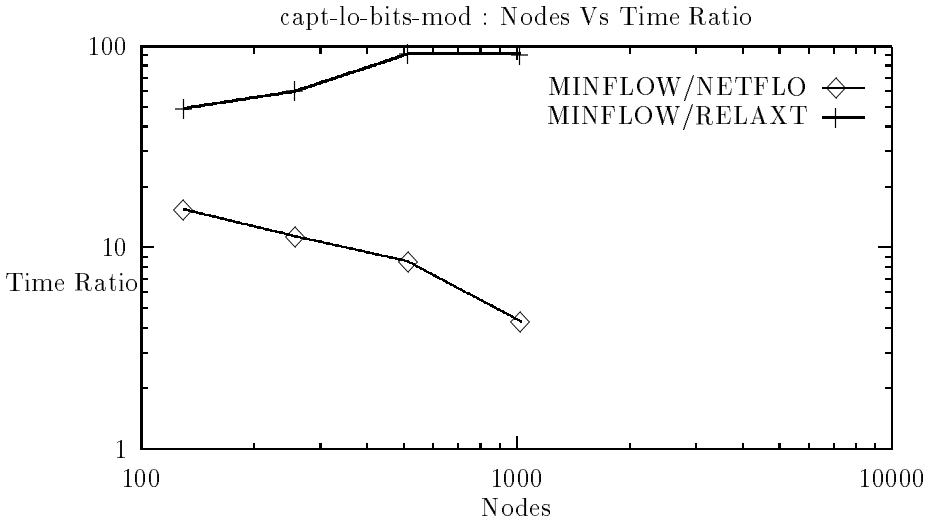


Figure 2.12: Statistics for family **capt-lo-bits-mod**

Size			Avg. Time (seconds)			Std. Dev. Time (seconds)		
Nodes	Arcs	#	MINFLOW	NETFLO	RELAXT	MINFLOW	NETFLO	RELAXT
130	2113	10	16.1	1.5	0.5	2.4	0.0	0.0
258	8321	10	81.5	11.7	2.0	10.3	0.4	0.0
514	33025	10	529.7	93.6	8.0	97.6	2.3	0.1
1026	131585	2	3983.9	771.6	32.2	452.3	1.7	0.0

Size			Iterations			Iterations (Std.dev)		
Nodes	Arcs	#	MINFLOW	NETFLO	RELAXT	MINFLOW	NETFLO	RELAXT
130	2113	10	97.1	1.8	0.2	24.2	0.4	0.4
258	8321	10	154.5	4.2	3.0	40.1	2.1	2.4
514	33025	10	351.9	20.0	17.6	56.1	5.9	5.6
1026	131585	2	1137.0	89.0	73.5	91.0	1.0	1.5

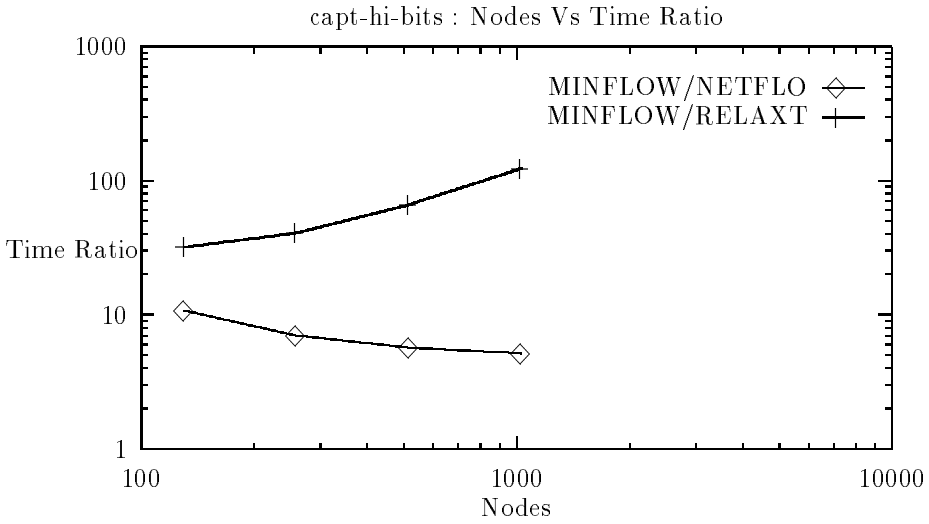


Figure 2.13: Statistics for family **capt-hi-bits**

Size			Avg. Time (seconds)			Std. Dev. Time (seconds)		
Nodes	Arcs	#	MINFLOW	NETFLO	RELAXT	MINFLOW	NETFLO	RELAXT
130	2113	10	26.3	1.6	0.6	3.0	0.0	0.0
258	8321	10	142.8	11.7	2.2	17.2	0.4	0.2
514	33025	10	895.9	94.3	8.7	60.9	2.4	0.0
1026	131585	2	3713.4	852.9	37.6	228.2	1.3	0.2

Size			Iterations			Iterations (Std.dev)		
Nodes	Arcs	#	MINFLOW	NETFLO	RELAXT	MINFLOW	NETFLO	RELAXT
130	2113	10	203.7	361.1	159.1	16.2	39.8	14.7
258	8321	10	278.4	758.9	322.9	21.1	46.9	23.1
514	33025	10	455.7	1525.1	657.2	14.3	67.2	33.7
1026	131585	2	594.5	3197.0	1234.5	5.5	75.0	24.5

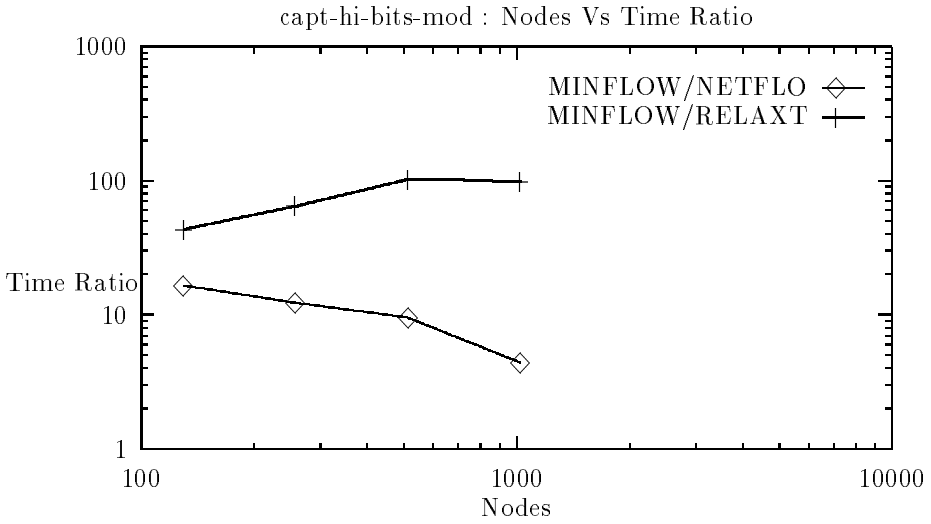


Figure 2.14: Statistics for family **capt-hi-bits-mod**

Size			Avg. Time (seconds)			Std. Dev. Time (seconds)		
Nodes	Arcs	#	MINFLOW	NETFLO	RELAXT	MINFLOW	NETFLO	RELAXT
256	512	3	9.8	0.4	0.2	0.1	0.0	0.0
529	1058	3	26.1	1.3	0.7	0.1	0.0	0.1
1024	2048	3	62.5	4.2	1.4	0.5	0.2	0.1
2116	4232	3	171.5	16.2	4.4	7.8	1.0	0.5
4096	8192	3	436.7	55.8	10.4	25.9	1.8	0.8
8281	16562	3	1199.1	239.8	27.3	54.0	2.0	2.8
16384	32768	3	3060.4	1046.9	78.0	98.4	27.2	8.0
33124	66248	3	8887.6	4667.0	169.1	90.9	138.3	26.2
65536	131072	3	24706.9	19616.1	540.2	182.2	1031.3	49.0

Size			Iterations			Iterations (Std.dev)		
Nodes	Arcs	#	MINFLOW	NETFLO	RELAXT	MINFLOW	NETFLO	RELAXT
256	512	3	219.3	1016.7	672.0	6.6	58.9	30.6
529	1058	3	282.3	2531.7	1760.3	6.8	90.7	205.5
1024	2048	3	355.7	5833.7	3110.3	5.4	138.0	215.3
2116	4232	3	527.3	13784.7	6949.3	2.6	469.3	194.3
4096	8192	3	751.3	28993.3	13129.7	12.0	1005.1	483.9
8281	16562	3	1115.7	67341.3	27472.7	26.4	499.2	282.2
16384	32768	3	1558.3	151026.0	57056.7	15.0	2766.8	640.7
33124	66248	3	2378.7	351469.7	113439.3	35.7	6161.2	1252.2
65536	131072	3	3483.3	779332.3	226449.3	33.2	8654.5	1996.9

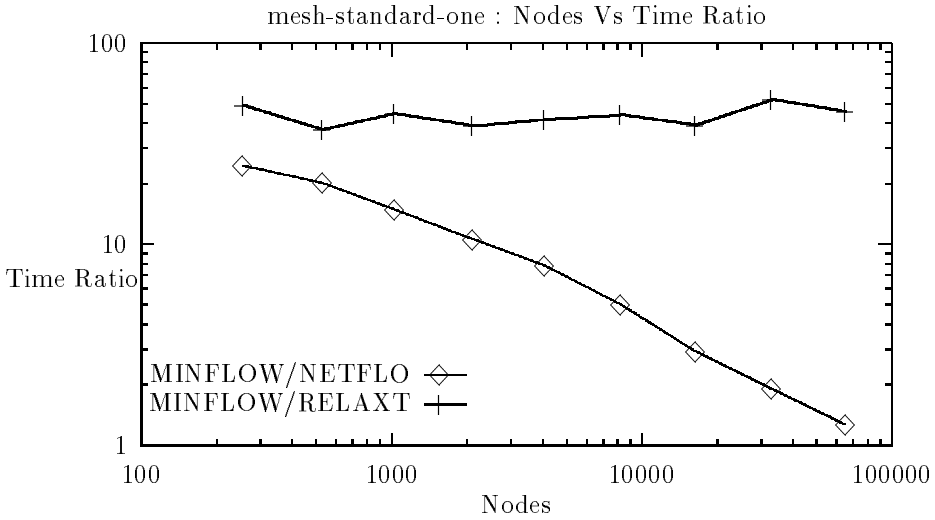


Figure 2.15: Statistics for family **mesh-standard-one**

Size			Avg. Time (seconds)			Std. Dev. Time (seconds)		
Nodes	Arcs	#	MINFLOW	NETFLO	RELAXT	MINFLOW	NETFLO	RELAXT
256	1024	3	16.7	0.9	0.4	0.4	0.0	0.0
529	2116	3	44.9	3.6	1.1	1.3	0.1	0.0
1024	4096	3	110.0	13.0	2.2	1.7	0.3	0.3
2116	8464	3	306.5	54.2	6.2	6.4	0.9	0.3
4096	16384	3	761.2	211.8	15.2	8.7	5.4	1.5
8281	33124	3	2035.9	930.4	46.5	14.9	6.1	4.3
16384	65536	3	5958.4	3892.7	100.3	39.2	20.4	21.4
33124	132496	3	16779.6	18285.5	346.3	553.8	1163.5	14.2
65536	262144	3	33900.3			23984.5		

Size			Iterations			Iterations (Std.dev)		
Nodes	Arcs	#	MINFLOW	NETFLO	RELAXT	MINFLOW	NETFLO	RELAXT
256	1024	3	269.3	3028.3	903.7	1.2	109.4	74.4
529	2116	3	377.0	8143.3	1979.7	5.0	281.3	162.1
1024	4096	3	530.0	18902.3	3614.7	11.3	180.9	414.5
2116	8464	3	764.0	44960.0	8056.0	5.7	795.4	348.5
4096	16384	3	1091.7	100814.0	17023.3	2.5	539.2	1135.2
8281	33124	3	1659.3	236127.3	38345.0	27.5	1227.2	367.5
16384	65536	3	2444.0	526127.0	71911.7	23.3	3452.9	3701.2
33124	132496	3	3751.3	1249134.7	156234.0	146.4	23765.4	2685.1
65536	262144	3	3626.3			2564.2		

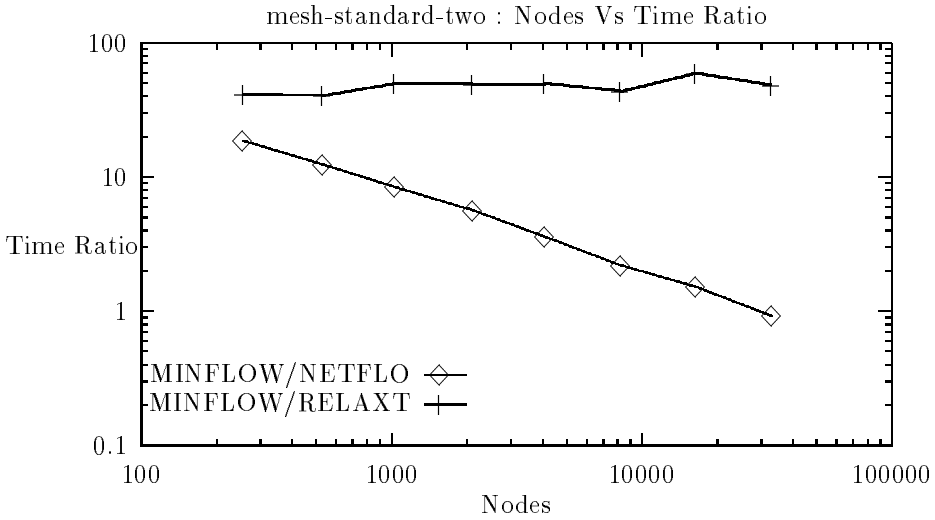


Figure 2.16: Statistics for family **mesh-standard-two**

Size			Avg. Time (seconds)			Std. Dev. Time (seconds)		
Nodes	Arcs	#	MINFLOW	NETFLO	RELAXT	MINFLOW	NETFLO	RELAXT
256	2048	3	27.7	1.9	0.7	0.4	0.0	0.0
529	4232	3	74.3	7.9	1.8	1.2	0.1	0.1
1024	8192	3	190.4	31.7	4.0	14.5	1.1	0.1
2116	16928	3	472.0	147.7	10.1	13.7	2.1	0.9
4096	32768	3	1143.2	605.5	29.1	26.6	11.8	3.1
8281	66248	3	3295.4	2736.6	67.8	102.2	57.1	2.2
16384	131072	3	9191.0	12084.4	182.8	180.4	92.7	4.2
33124	264992	3	18096.9			12796.4		

Size			Iterations			Iterations (Std.dev)		
Nodes	Arcs	#	MINFLOW	NETFLO	RELAXT	MINFLOW	NETFLO	RELAXT
256	2048	3	301.3	6717.7	975.0	17.2	158.1	82.0
529	4232	3	414.3	19854.0	2092.0	4.8	505.1	193.8
1024	8192	3	625.7	50809.7	4354.0	12.7	1087.9	141.4
2116	16928	3	890.0	126310.3	10060.7	21.5	758.8	603.5
4096	32768	3	1265.0	286119.0	20588.3	28.3	3267.3	2547.8
8281	66248	3	1993.0	664587.7	42485.0	39.9	3579.5	423.3
16384	131072	3	2974.0	1500231.7	87171.7	32.5	11059.1	2153.2
33124	264992	3	3084.0			2180.8		

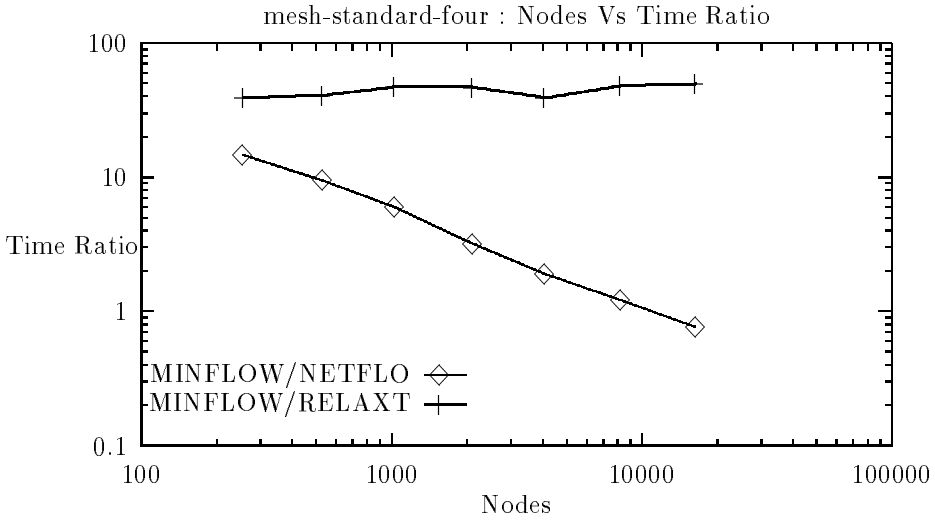


Figure 2.17: Statistics for family **mesh-standard-four**

Size			Avg. Time (seconds)			Std. Dev. Time (seconds)		
Nodes	Arcs	#	MINFLOW	NETFLO	RELAXT	MINFLOW	NETFLO	RELAXT
256	4096	3	51.4	3.6	1.5	2.5	0.1	0.0
529	8464	3	132.1	16.4	3.5	2.4	0.3	0.1
1024	16384	3	327.2	66.6	7.4	16.6	1.0	0.4
2116	33856	3	801.0	347.3	19.5	18.5	5.7	0.2
4096	65536	3	2148.2	1651.9	48.9	56.9	35.3	3.2
8281	132496	3	5160.2	6977.0	116.6	230.6	35.5	6.0
16384	262144	3	10374.1			7363.3		

Size			Iterations			Iterations (Std.dev)		
Nodes	Arcs	#	MINFLOW	NETFLO	RELAXT	MINFLOW	NETFLO	RELAXT
256	4096	3	307.7	12141.3	1152.0	15.3	470.2	84.2
529	8464	3	470.0	41434.0	2794.0	19.6	921.1	211.6
1024	16384	3	649.3	114986.0	5397.3	17.4	1190.6	125.2
2116	33856	3	933.3	309586.0	11669.7	13.3	6483.7	472.8
4096	65536	3	1363.0	711413.3	24505.0	26.2	9406.9	1226.1
8281	132496	3	1983.7	1672519.7	50600.7	42.6	986.2	1407.7
16384	262144	3	2154.3			1526.1		

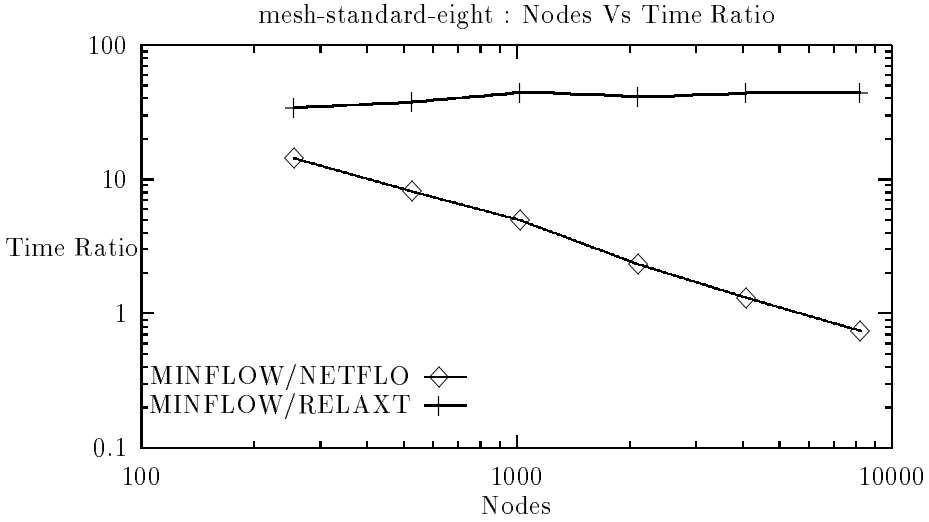


Figure 2.18: Statistics for family **mesh-standard-eight**

Chapter 3

Practical Cycle Borrowing Analysis Using Interior Point Algorithms

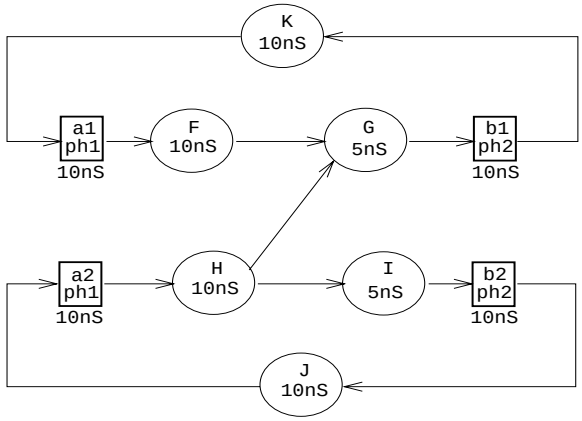
Circuits designed with level-sensitive latches exhibit the benevolent phenomenon of *cycle borrowing,* a topic of considerable investigation [13] [41] [42] [27]. [26]. In order to properly exploit cycle borrowing, the designer requires an efficient and accurate timing analysis. Up to now it has been possible to be fast, or to be accurate, but not both together. The most accurate method is based on formulating the analysis entirely as a linear program [13], but for exactly that reason the analysis runtime would ordinarily degrade rapidly in the size of the circuit and therefore this approach has not been seriously pursued.

In this paper we report the first truly practical cycle borrowing analysis technique, based entirely on a linear programming formulation. The method reported here, as best we are aware, is the first to concisely state the proper objective function to be used for the underlying linear program. Then, the method takes into account the special properties inherent in cycle borrowing linear programs, by utilizing a fast interior point algorithm that judiciously combines a new convex programming algorithm [44] with a minimum cost flow algorithm [30]. The technique has been implemented as the CONFLO timing analyzer and tested on real-sized latch-based circuits. We report timing analysis results, confirm their accuracy where possible, and state workstation execution times in terms of both size of the circuit and size of the derived linear program. For designs which once took us several hours to analyze, CONFLO now provides the same results in minutes.

3.1 Definitions and Previous Work

A valuable introduction to cycle borrowing is found in [13]. Besides introducing the important concepts of *retardation* and of clock waveform *timing intervals* (which we designate L_i), [13] also states that the task of assigning values to the L_i such that their sum is minimal can be formulated as a linear programming problem. Having stated this, [13] dismisses the linear programming formulation as "very costly" and an alternative convergence technique for solving the L_i's is proposed. The inaccuracy of that proposed method was later demonstrated by [41].

After introducing a different model for the clock cycle and an analysis particular to that model, [41] also concede the problem's relationship to linear programming. However, they arrive at a hybrid computation in which an initial linear programming computation provides an optimal solution to the cycle times, but only tentative solutions to the retardation values. These tentative retardation values are then supposedly corrected by a subsequent (reportedly quickly converging) iteration which is called "sliding." Correct retardation values are critical to circuit design, since these are needed to identify circuit slacks and to guide optimizing transformations.



solution $min \sum L_i$	**45.0 nS**
L_0 **(ph.1 high)**	**20.0 nS**
L_1 **(both low)**	**0.0 nS**
L_2 **(ph.2 high)**	**25.0 nS**
L_3 **(both low)**	**0.0 nS**
D_{a1}	**10.0 nS**
D_{a2}	**10.0 nS**
D_{b1}	**15.0 nS**
D_{b2}	**15.0 nS**

Figure 3.1: Linear program Minimizing the Cycle Width Only

The above Linear program of the cycle borrowing analysis only minimizes the cycle width. Observe that retardation values D are consistent but incorrect.

Unfortunately this partial departure from a direct linear programming solution is dangerous as we now show by example. We solve the linear program for the circuit of Figure 3.1 as specified by [41]. Ignoring skew, setup and hold effects in the spirit of the simplified examples found in both [13] and [41] (we discuss setup and hold below) we obtain the solution shown in Figure 3.1. The retardation values D_j are then subjected to the "sliding" procedure. That procedure will be unable to locate any deviation upon which to iterate, and so it will declare the D_j values correct. But they are not correct; the correct solutions for the D_j are shown in Figure 3.2. One can derive the correct values intuitively by adding a small increment ϵ to each of the L_i and then simulating the circuit starting with the original "uncorrected" D_j's. The ϵ causes the actual retardations to shrink during each subsequent cycle. This shrinking halts when the D_j's come firmly up against the synchronizing action of the latches. Indeed, that is the very intuition behind "sliding," but that intuition does not translate into a correct method of computation.

However, the correct values for D_j are immediately obtained from a linear programming formulation as follows:

Obj 1. Minimize $\sum L_i$ subject to the set of linear programming constraints.

Obj 2. Retaining the minimum value of L_i from Obj 1, and subject to the same constraints as during Obj 1, minimize $\sum D_j$.

In practice it is possible to combine both steps into a single linear program by proper formulation of the objective function over L and D, as we will show in Section 3.3.

Linear programming constraints to deal with setup and hold times are readily generated, rather than having to appeal to *ad hoc* post-linear-programming corrections as in [42]. Since no previous work seems to recognize this, we will expound this in detail here. Our model for latched circuits as found in [29] leads to a linear program having timing intervals L_i and retardations D_j as variables, exactly as anticipated by [13]. Now consider a pair of latches, an "upstream" latch which opens in interval k at time $start(k)$ and a latch "downstream" from it which closes in interval $k + n$. The time at which the path from the upstream latch to the input of the downstream latch becomes stable is

D_{a1} **0.0 nS**
D_{a2} **0.0 nS**
D_{b1} **5.0 nS**
D_{b2} **5.0 nS**

Figure 3.2: Correct Retardation Values D

$$in_stable = start(k) + D_{up} + path_length$$

where *path_length* is the *longest* path between the latches, and where we ignore clock skew for simplicity. Since the downstream latch closes at $start(k+n)$ we require

$$in_stable + setup_time \leq start(k+n)$$

which we can rewrite as

$$in_stable + setup_time \leq start(k) + \sum_{i=k}^{k+n-1} L_i$$

so that $start(k)$ cancels from both sides giving

$$D_{up} + path_length + setup_time \leq \sum_{i=k}^{k+n-1} L_i,$$

which is a simple linear programming constraint in terms of D_{up}, L's, and constant terms determined by static circuit path analysis. One such constraint is generated for each such pathwise-connected pair.

Hold times can be similarly dealt with: the earliest time that the input to a downstream latch can change, due to an upstream latch opening in interval k is

$$in_change = start(k) + path_length$$

where *path_length* is now the *shortest* path between the latches.[1] We can rewrite *in_change* as

[1] There is a pessimism here due to not including a term for the *minimum* retardation at the upstream latch, denoted d_j in [42]. Here we assume this term to be zero.

$$in_change = start(k-n) + \sum_{i=k-n}^{k-1} L_i + path_length$$

With the downstream latch closing in interval $k-n$ we require

$$in_change \geq start(k-n) + hold_time$$

where again $start(k-n)$ cancels, leaving a constraint in L's. One such constraint is emitted for every connected pair.

Due to space considerations we cannot expound the "complete" [42] linear programming formulation of the cycle borrowing problem here, but hope that the preceding examples have convinced the reader that it is actually a straightforward derivation.[2] Furthermore we have given evidence about the dangers involved in departing from a pure linear programming formulation. However, in order to retreat to the safety of the linear programming formulation we need to have a practical, computationally efficient technique which exploits the particular properties inherent in the underlying linear program, and *that* is what the present paper is all about.

3.2 Generating the Linear Program from the Circuit

In this paper we concentrate on the timing *synthesis* problem for two-phase non-overlapping clocking, that is, to find the minimum values for L_i having been given the circuit, as in Figure 3.1. The corresponding timing *verification* problem simply fixes the L_i values by equality constraints and the resulting linear program is checked for feasibility.

The constraints are generated by an automatic analysis of the circuit. This has linear time path search complexity as discussed in [29] [28]. Each constraint deals with at most two latches, as was the case with the setup and hold constraints given above. For an upstream latch opening in interval k toward a downstream latch opening in interval $k+n$ we have (using the longest path between them)

$$\sum_{i=k}^{k+n-1} L_i \geq D_{up} + path_length - D_{dwn}.$$

[2]The complete problem includes correction terms for skew in both rising and falling clock events, equations and corrections dealing with short paths and the *minimum* retardations d_j, additional concerns for the control signal to gated latches, relaxation on the L_i's to allow individual members to become negative, and many other details.

where the index i is computed modulus the number of epochs, a simple and elegant mechanism by which we "cross the cycle boundary." For an upstream latch opening in interval k toward a downstream latch *closing* in interval $k+n$ we require

$$\sum_{i=k}^{k+n-1} L_i \geq D_{up} + path_length$$

where $path_length$ includes the path through the downstream latch and thus (if the upstream and downstream latches are one and the same) also gives us latch pulse width constraints. We also use the less obvious structural upper bounds on the retardations,

$$D_j \leq source_path_length$$

where $source_path_length$ is the longest path from all closed latches to the latch of discourse during the interval in which that latch opens. The usefulness of these constraints is discussed in Section 3.3 below. Clearly, each constraint contains at most two nonzero D coefficients, always having value 1 or -1. This is an important property of the cycle borrowing problem, having negative consequences when using the simplex method [43][3] but positive consequences in our present solution method, as discussed next.

3.3 Analysis Methods

We first implemented a naive simplex method [24] modifying the management of the objective function so that, in addition to the symbolic "big M" and direct cost coefficient, we had a third coefficient which we called "little S," an infinitesimal which only became significant during pivoting when the other compared costs were equal. We assigned the cost $1 + 0\,S$ to each L_i and the cost $0 + 1\,S$ to each D_j. The result is that the procedure drives L_i to a minimum and then, maintaining the minimum sum over L, forces the D_j's to their minima. We called this procedure SYMSIM, for "symbolic simplex," and tried to apply it to the benchmark circuits to be described in Section 3.4. As [13] had fully anticipated, this was "very costly," having running times in hours for modestly-sized circuits. This led to the search for other linear programming solution methods.

[3] Pg.141, "The Worst-Case Behavior of the Simplex Method."

The underlying linear program may be rewritten as

Problem P:

$$\max \quad M \sum_j y_j - \sum_i L_i - S \sum_k D_k$$

$$\text{Subject to} \quad A^T D + y \leq c(L)$$

$$\text{and} \quad y \leq 0$$

where D is the vector of the retardation variables (D_k's), $c(L)$ is a vector valued linear function of L_i's, A^T is the matrix of coefficients of the D_k's and y_j's are the slacks. Note that the "big M" drives the slacks y to zero. Each row of A^T contains either two non-zeros which are $+1$ and -1, or a single non-zero which is a ± 1. An interesting property of *Problem P* is that for a fixed L it reduces to the dual of a minimum cost flow problem [36]. This enables us to solve *Problem P* by a judicious combination of a new interior point convex programming algorithm [44], and a fast solver for minimum cost flow problems [30] for generating gradients.

Let

$$F(L) = \max \quad M \sum_j y_j - S \sum_k D_k$$

$$\text{Subject to} \quad A^T D + y \leq c(L)$$

$$\text{and} \quad y \leq 0$$

Let

$$\phi(L) = F(L) - \sum_i L_i$$

Then in the space of L_i's, *Problem P* may be expressed as

Problem C:

$$\text{Find an } L \text{ that maximizes } \phi(L)$$

Since $F(L)$ is a piecewise linear concave function, so is $\phi(L)$.

Let L^* denote the solution to *Problem C*. An optimal set of D's may be computed by solving

$$\max \quad M \sum_j y_j - S \sum_k D_k$$

$$\text{Subject to} \quad A^T D + y \leq c(L^*)$$

$$\text{and} \quad y \leq 0$$

We solve *Problem C* using a new convex programming algorithm [44] which works by encapsulating the optimum in a polytope P and successively cutting the polytope through its center ω by a tangent plane to the level surface of $\phi(L)$ passing through ω. Since $\phi(L)$ is concave, it is guaranteed that if P contains the optimum, then so does the smaller polytope $P' = P \cap \{L : \nabla\phi(\omega)^T L \geq \nabla\phi(\omega)^T\omega\}$ where $\nabla\phi(\omega)$ denotes the gradient of ϕ evaluated at ω. Each step consists of computing the center of the current polytope P, finding the gradient $\nabla\phi(\omega)$ and resetting P to the smaller polytope P'. The volume of P shrinks geometrically and a point close enough to the optimum L^* is isolated once P is sufficiently small.

Note that $\nabla\phi(L) = -\nabla(\sum_i L_i) + \nabla F(L)$. $\nabla F(L)$ may be evaluated by solving a minimum cost flow problem which is the dual of the linear program defining $F(L)$. In particular, let $x_{opt}(\hat{L})$ be the solution to

	combinational component count	latch count
H 2910	269	169
J 2910	227	172
U 2910	238	150
RISC	973	362
RS DEC	3256	704

Figure 3.3: Number of components in Test Circuits

$$\max c(\hat{L})^T x$$

$$\text{Subject to} \quad Ax = 0$$

$$\text{and} \quad 0 \leq x \leq M$$

Locally around $\hat{L}$, $F(L)$ equals the linear function $x_{opt}(\hat{L})^T c(L)$ and $\nabla F(L)$ may be obtained by differentiating this linear function.

In summary we solve *Problem P* by a convex programming algorithm executed in the space of L's which has low dimension. A minimum cost flow problem is solved at each step to generate the gradient of $\phi(L)$. The solution of these flow problems is the most time consuming part and we do it using a fast interior point algorithm that was developed in [30]. Exploiting the network type structure of the problem and the structural upper bounds on the retardation variables (D's) together with a judicious combination of algorithms gives speeds orders of magnitude better than the simplex method.

3.4 Test Circuits and Results

	no. of variables	no. of constraints	($\sum L$) and CPU time, symmetrical waveforms	
			SYMSIM	CONFLO
H 2910	154	1110	130 nS in 6 hrs	130 nS in 130 sec
J 2910	164	884	140 nS in 3.5 hrs	140 nS in 114 sec
U 2910	142	795	152 nS in 5 hrs	152 nS in 91 sec
RISC	323	2005	–	700 nS in 182 sec
RS DEC	701	2968	–	500 nS in 630 sec

Figure 3.4: Computational Results for Solutions Restricted to Symmetrical Waveforms

	no. of variables	no. of constraints	($\sum L$) and CPU time, unrestricted waveforms	
			SYMSIM	CONFLO
H 2910	154	1110	130 nS in 7 hrs	130 nS in 201 sec
J 2910	164	884	140 nS in 7.5 hrs	140 nS in 182 sec
U 2910	142	795	145 nS in 5.5 hrs	145 nS in 136 sec
RISC	323	2005	–	389 nS in 401 sec
RS DEC	701	2968	–	500 nS in 1095 sec

Figure 3.5: Computational Results for Unconstrained Waveforms

We then tested both SYMSIM and CONFLO on real circuits which had been developed on the BBDS register transfer level design system [8] by students of the VLSI Architecture course at Lund University. We selected the five largest designs which were available, and these are listed in Figure 3.3 as "H2910," "J2910," "U2910" (all of which are variants of the AM 2910 controller), "RISC" (an eight-bit RISC processor) and "RSDEC" (a Reed-Solomon error-correcting decoder). These were then mapped by MIS-II [7] onto the Actel field-programmable gate array component library. Each component is considerably more complex than a single Boolean logic gate, which should be kept in mind when studying the "component count" numbers in Figure 3.3.

The above Figure 3.4 gives the results of cycle borrowing analyses, and CPU time required to solve the resulting linear program, on the circuits of Figure 3.3. The first pair of columns shows the size of the resulting linear program to be solved. The second pair of columns in shows the solutions restricted to symmetrical waveforms, that is, phase 1 high time equal to phase 2 high time. The above Figure 3.5 gives the results of cycle borrowing analyses, and CPU time required to solve the

resulting linear program, on the circuits of Figure 3.3. The first pair of columns shows the size of the resulting linear program to be solved. The second pair of columns shows the solutions for unconstrained clock waveforms.

The computed minimum clock cycle times, together with the amount of CPU time needed to compute each result on a Sun SPARC workstation, are shown in Figures 3.4 and 3.5. The number of variables and constraints in the linear program resulting from each circuit is also given. The number of variables in each case is smaller than would be expected from the number of latches stated in Figure 3.3 due to the way we presently handle primary circuit inputs. In Figures 3.4 and 3.5 we actually solve two problems for each circuit. One is the synthesis of *symmetrical* waveforms, which means that the phase 1 clock high-time has the same period as the phase 2 clock high-time, while the two clock-off periods are also equal to each other. The other synthesis problem is without any such symmetry constraints. Each problem (the symmetrical and the unsymmetrical) is solved twice: once by the SYMSIM tool (where feasible) and once by the CONFLO tool. This allowed us to verify the results of one technique against the other.

We observe an improvement of two to three orders of magnitude in the running times. The growth rate of the running time of CONFLO is experimentally observed to be somewhere between $m \log m$ and $m\sqrt{n}$, where m is the number of constraints and n is the number of variables. Note that the analysis in [44, 30] guarantees that it is polynomial in m and n in the worst case.

3.5 Summary and Conclusions

The combination of a new convex programming algorithm and a network flow algorithm now makes it possible to solve very large cycle borrowing timing analysis problems accurately. This new algorithm is orders of magnitude faster than the simplex method thereby making it practical to solve real-sized timing problems on a workstation which had previously been abandoned as being infeasible.

Chapter 4

An Efficient Algorithm for Special Two-Variable Convex Programming Problems

4.1 Introduction

In this chapter, a linear time algorithm for special convex programming problems in two-variables with linear objective function is presented. We use a geometrical approach for the special problem whose feasible set is convex with certain restrictions on the form of the constraints.

Special Convex Programming Problems A two-variable special convex programming problem is defined as

$$\text{Min} \quad c^T x \quad \text{subject to} \quad g_i(x) \leq 0; \quad 1 \leq i \leq n \tag{4.1}$$

where $x \in \Re^2$ and $\{x : g_i(x) \leq 0\}$ is a convex set for $i = 1, \cdots, n$. We impose a further restriction that the number of intersections between the boundaries of every pair of distinct sets

$\{x : g_i(x) \leq 0\}$ and $\{x : g_j(x) \leq 0\}$, $i \neq j$

is less than or equal to a constant k. Let us refer to this restriction as **Restriction 1**.

The algorithm in this chapter is closely modeled after the linear programming algorithm in $\Re^2$ and $\Re^3$ using the technique of prune-and-search developed by [14] and independently by [35]. A tutorial description of the algorithm appears in [15].

This chapter is organized as follows. In section 4.2, some notations used in the rest of the chapter are presented. In section 4.3, some technical theorems required in the analysis of the algorithm are presented. In section 4.4, the algorithm is described. In section 4.5 we show that this algorithm works in time linear in the number of the constraints n, provided there are two oracles which work in constant time. In section 4.6, we give example **Oracles 1** and **2** from the theory of plane algebraic curves.

4.2 Notations and Definitions

We use the following definitions in this chapter.

1 Turning Point: A turning point of a boundary is defined to be the point where it has either a vertical or a horizontal tangent.

2 Rotation: Note that it is possible to apply a rotation to $\Re^2$ such that the objective function is horizontal and is defined by the new cost vector

$$c = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

For the rest of this chapter, we assume that the cost vector is in the above form.

3 Oracle 1: Given a set $\{x : g_i(x) \leq 0\}$, this oracle finds all the points at which the boundary of this set has vertical tangents. If we have an oracle that can find all the turning points, it can be used as **Oracle 1**.

4 Oracle 2: Given two distinct sets $\{x : g_i(x) \leq 0\}$ and $\{x : g_j(x) \leq 0\}$, this oracle finds all the intersections between the boundaries of these sets.

5 Sets S^+ and S^-: The boundary of each convex set $\{x : g_i(x) \leq 0\}$ can be expressed as a curve in two variables ξ_1 and ξ_2. Let this boundary be known as C_i. Each C_i can be divided into two segments, one segment whose normal has a slope $\in (\pi, 2\pi)$ and the second segment whose normal has a slope $\in (0, \pi)$. Let C_i^+ and C_i^- be the first and second segments respectively. The boundary of any convex set $\{x : g_i(x) \leq 0\}$ can thus be divided into two segments C_i^+ and C_i^-. Define

$$S^+ = \bigcup_{i=1}^{n} C_i^+$$

$$S^- = \bigcup_{i=1}^{n} C_i^-$$

6 Sets C^+ and C^-: Define

$$C^+ = \bigcap_{i=1}^{n} C_i^+$$

$$C^- = \bigcap_{i=1}^{n} C_i^-$$

7 Data D: The data of the convex program under consideration is defined as

$$D = S^+ \cup S^-$$

8 Range Σ: The range of the convex program under consideration Σ is defined as a vertical strip which contains the optimal point. In other words, Σ is the region between two vertical lines $l_{left} : \xi_1 = a_{left}$ and $l_{right} : \xi_1 = a_{right}$. $a_{left} \leq a_{right}$ when the program is feasible.

4.3 Some Theorems

We present some theorems that are useful in proving the desired time bounds of the algorithm presented herein.

1. Every convex set has at most four turning points.

Proof: Without loss of generality, let us look at a vertical tangent that is to the left of the convex set. By way of contradiction assume that there are two such tangents, at points p and q. We also assume that the boundary does not have any degeneracies (*i.e.* the boundaries of all sets $\{x : g_i(x) \leq 0\}$ are irreducible) and the points p and q are in general position (*i.e.* they do not have the same ξ_1 coordinate). The theorem on the existence of a supporting hyperplane (see [17]) says that the convex set should lie entirely to one side of the supporting hyperplane. But this results in a contradiction as p that is to the left of point q yet belongs to the convex set and should be to the right of point q. □

2. If the two boundaries C_i^+ and C_j^+ (similarly C_i^- and C_j^-) corresponding to the sets $\{x : g_i(x) \leq 0\}$ and $\{x : g_j(x) \leq 0\}$ contain no intersections in the current range Σ then one of the two constraints associated with these two boundaries is redundant and can be eliminated.

Proof: Look at segments C_i^+ and C_j^+(similarly C_i^- and C_j^-). Since there are no intersections between these two boundaries in the range Σ, they do not cross each other within the range. This implies that either the convex set $\{x : g_i(x) \leq 0\}$ contains the convex set $\{x : g_j(x) \leq 0\}$ or vice versa. The constraint corresponding to the containing convex set is redundant and can be eliminated. □

4.4 A Linear Time Algorithm

In this section we present the algorithm and prove the claimed time bounds. The description of our algorithm follows closely the description of [14, 35] as given in [15].

We start with a range that includes the feasible set of the convex program. Next the range of the convex program is decreased by placing several vertical test lines and solving the resulting one-dimensional convex programs which are limited to these test lines. We reduce the size of D by applying **Theorem 2** on the new range and D. At this point, we are ready to apply the algorithm recursively to a convex program which has a smaller number of constraints compared to the original. We show that the size of the new set D decreases geometrically which gives rise to a recurrence relation of the form

$$T(n) = T(n/p) + O(n)$$

where $T(n)$ is the number of operations to solve the convex program, n is the number of constraints, and n/p is the number of constraints in the new convex program where $p > 1$. The solution to this recurrence relation is $O(n)$ (see [12]).

4.4.1 Psuedocode Description of the Algorithm

Below we use the same terminology as used in [15]. D stands for the set of data and Σ stands for the range which is known to contain all solutions.

Algorithm for Convex Program(Prune-and-search)

Preprocessing: See subsection 4.4.2.

if the size of D is at most some constant **then**

Use a trivial procedure to solve the problem.

else

SEARCH: Iterate the following two steps some constant number of times:

FIND-TEST: Find an appropriate test line t.

REDUCE-RANGE: Decrease the range Σ which contains all solutions by answering the test around test line t.

PRUNE: Eliminate some subset of D which is irrelevant in Σ.

RECUR: Repeat the computation for the new sets D and Σ.

endif

4.4.2 Detailed explanation of the Algorithm

Preprocessing:

1. Apply a rotation to $c^T x$ and all the convex sets $\{x : g_i(x) \leq 0\}$ so that the cost vector c becomes vertical pointing downwards (*i.e.* level surfaces of the objective function are horizontal). This can be done in time $O(n)$.

2. Use **Oracle 1** on the boundary of each convex set $\{x : g_i(x) \leq 0\}$. If **Oracle 1** takes constant time then the time required in this step is $O(n)$.

3. Compute sets S^+ and S^-. Set $D = S^+ \cup S^-$. Initial range is set by taking the maximum of the ξ_1 coordinates of the points of the boundaries where there is a vertical tangent to the right of the boundary and making this a_{left}. Similarly a_{right} is the minimum of the ξ_1 coordinates of the points of the boundaries where there is a vertical tangent to the left of the boundary. This step also takes time $O(n)$.

So, preprocessing takes time $O(n)$ to complete.

PRUNE: Eliminate Redundant Constraints

This step involves eliminating those constraints from the pairs satisfying conditions set down in **Theorem 2**. For the algorithm to work in time $O(n)$ all we have to do is to guarantee that a fraction of the constraints are eliminated in this step. This guarantee is obtained in **FIND-TEST** step. **PRUNE** is of time $O(n)$.

REDUCE-RANGE: Decrease the Range of the Convex Program

This step is slightly more complicated but takes $O(n)$ time. In this step, we need to decide, given a vertical test line $t : \xi_1 = \tau$, which side of the test line t the solution to our convex program lies, or whether the test line t itself contains the solution. To do this, we first compute the intersection points

$$p^+(t) = t \cap C^+ \quad \text{and} \quad p^-(t) = t \cap C^-$$

We have to keep in mind that the sets C^+ and C^- are implicitly given as a set of unordered boundaries by the convex sets $\{x : g_i(x) \leq 0\}$. We can use **Oracle 2** to compute $p^+(t)$ and $p^-(t)$. This can be done in $O(n)$ time, provided that **Oracle 2** takes constant time. If $p^+(t)$ lies above $p^-(t)$ then t does not contain any feasible point. If $p^+(t) = p^-(t)$ then we report that the solution is $p^+(t)$. If $p^+(t)$ lies below $p^-(t)$, then $p^+(t)$ is a feasible solution.

Next, we want to find out whether the solution improves if we move to the left or right of t. By an "improved" solution we mean that either the infeasibility decreases in case of infeasibility or the objective function gets smaller in case of feasibility. Because of convexity, the solution can improve on only one side of the test line t since t is a separating hyperplane for the two-variable convex program under consideration. To decide which side of the test line t we need to move next, we place two test lines t' and t'' arbitrarily close to the left and to the right of t. After finding the

points $p^-(t')$, $p^+(t')$, $p^-(t'')$, and $p^+(t'')$, it is easy to decide whether the solution improves to the left or right of t or t itself contains the solution.

FIND-TEST: Determine the Test Line

Designing this step is crucial in getting the linear time bound for the algorithm. The important point to remember while finding the test line is to get at least a fraction of the constraints to be eliminated in the **PRUNE** step. The idea is to pair the boundaries of the sets $\{x : g_i(x) \leq 0\}$ first and choose the test line t depending on the location of the intersections between matched boundaries. As a first step, we match each but possibly one boundary from S^+ with another boundary from S^+ and analogously with boundaries in S^-.

Let us assume that two boundaries which have been paired intersect in at most a constant number of points k. Since we started with n boundaries, we have $n/2$ pairs and the total number of intersections are $kn/2$. Let us find all the $k+1$st quantiles of the ξ_1-coordinates of these intersection points and place k vertical test lines at these points. This can be done in time $O(n)$ as long as k is a constant (see [12]). It is easy to apply the **REDUCE-RANGE** step at each of these k test lines. This will reduce the range to one of the k intervals. Since these are $k+1$st quantiles, each of these intervals cannot contain more than $\frac{1}{k+1}.\frac{kn}{2}$ intersections. That means the range cannot contain more than $\frac{k}{k+1}.\frac{n}{2}$ intersections between the pairs. We conclude that there are a fraction of the pairs (namely $\frac{1}{k+1}.\frac{n}{2}$) whose intersections lie wholly outside the range. By **Theorem 2**, we can eliminate $\frac{1}{k+1}.\frac{n}{2}$ constraints (namely, one from each pair not intersecting in the range).

4.5 Analysis

As we argued in section 4.4 at most $\frac{1}{k+1}.\frac{kn}{2}$ of the intersections between the matched pairs are in the range Σ. This means that we can eliminate a fraction of the constraints in the **PRUNE** step. Let us compute the number of eliminated constraints.

The total number of pairs that have their intersections outside the range are at least m where

$$m = \frac{n}{2} - \frac{1}{k+1}.\frac{kn}{2} = \frac{1}{k+1}.\frac{n}{2}$$

This guarantees the elimination of at least m constraints in the **PRUNE** step. For at most a constant number of intersections k between every two boundaries of sets $\{x : g_i(x) \leq 0\}; i = 1, \cdots, n$, we eliminate $\frac{n}{2(k+1)}$ constraints as redundant in the **PRUNE** step which gives us the following recurrence relation for the number of operations $T(n)$.

$$T(n) = T(\frac{2k+1}{2k+2}n) + O(n)$$

$T(n) = O(n)$ since the argument of function T decreases geometrically. Also see [12].

4.6 Discussion

In case of plane algebraic curves, we can use the computation of zeros of **Resultants** and **Discriminants** for **Oracles 1** and **2** (see [9]). These methods can be applied when the boundaries of the convex sets $\{x : g_i(x) \leq 0\}$ are polynomials of at most degree k.

It is possible to relax **Restriction 1** without changing the complexity of the algorithm, provided that the total number of intersections among the boundaries of all convex sets $\{x : g_i(x) \leq 0\}$, $i = 1, \cdots, n$ is $O(n)$. We choose the constant k for finding the quantiles in **FIND-TEST** step of the algorithm to be greater than the constant hidden in the "big oh" notation of $O(n)$. Then the analysis of section 4.5 applies.

Chapter 5

Analysis of SPDDD Linear Systems Whose Graphs Have Restricted Topologies

5.1 Motivation

The solution of Symmetric Positive Definite Diagonally Dominant (SPDDD) linear systems is important because these kinds of matrices arise in several application areas. Some of these areas are listed below.

Scalar Second Order PDEs When continuum problems in Heat Transfer, Electrostatics, Electromagnetics, Ground Water Flow, and Diffusion (with or without reaction) are discretized using finite differences and certain kinds of finite elements, they give rise to SPDDD linear systems. The discretization also naturally relates a graph with the SPDDD matrix in the linear system.

Discrete Problems This class includes network flows, resistive networks and laminar flows in pipe networks. The last problem can be cast as a resistive network problem [11]. The solution to these problems involves solving an SPDDD linear system. Since the problems themselves are defined on a graph, this graph is naturally associated with the SPDDD matrix in the linear system.

In this chapter, we analyze the quality of a new graph-based preconditioner for SPDDD matrices whose associated graphs have restricted topologies. Initially we carry out the analysis in detail for a $\sqrt{n} \times \sqrt{n}$ grid graph $G_{\sqrt{n}}$. This graph would serve as the running example throughout this chapter. At the end of the chapter, we will give some more topologies for which the analysis would apply unmodified.

5.2 Certain SPDDD Matrices and Associated Graphs

In this section, we describe how to obtain a graph associated with an SPDDD matrix and vice versa.

Let $A = [a_{ij}], 1 \leq i, j \leq n$ be a Symmetric Positive Definite Diagonally Dominant matrix whose off-diagonal entries are negative or zero. Since A is diagonally dominant

$$a_{ii} \geq \sum_{1 \leq j \leq n, j \neq i} |a_{ij}|$$

Let Δ_i be the diagonal excess of row i of A and Δ be a diagonal matrix with the diagonal entries $\Delta_i, 1 \leq i \leq n$. Let ε_{ij} be a vector in $\Re^n$ such that $\varepsilon_{ij}[i] = +1$ and $\varepsilon_{ij}[j] = -1$ (other entries in ε_{ij} are zero). Then A can be written as

$$A = \Delta + \sum_{a_{ij} \neq 0, i<j} |a_{ij}| \varepsilon_{ij} \varepsilon_{ij}^T \tag{5.1}$$

We can define the associated graph $G(\mathrm{A}) = (V, E)$ as follows. With each vertex $v_i \in V$ we associate a weight which is the diagonal excess Δ_i of A. There is an edge e_{ij} between two vertices i and j iff $a_{ij} \neq 0$. We associate a weight $|a_{ij}|$ with edge e_{ij}.

It is easily seen that given a graph $G(\mathrm{A}) = (V, E)$ with the associated vertex and edge weights, we can derive the associated SPDDD matrix A whose off-diagonals are negative or zero and vice-versa. Intuitively, the SPDDD matrix A with negative or zero off-diagonal entries corresponds to a resistive network. A negative off-diagonal entry a_{ij} of A can be thought of as a conductance of value $|a_{ij}|$ between vertices i and j in the resistive network. The diagonal excess Δ_i can be thought of as a conductance of value Δ_i between vertex i and an external vertex called the ground node.

Since an SPDDD matrix with negative or zero off-diagonal entries is equivalent to a resistive network, we call such matrices resistive SPDDD matrices. Because of this one-to-one correspon-

dence between a resistive SPDDD matrix A and its associated graph $G(\mathrm{A})$ with vertex and edge weights as defined above, we use A and $G(\mathrm{A})$ interchangeably in this chapter.

At first it might seem that the condition *the off-diagonal entries must be negative or zero* is very restrictive. This is not the case because all problems described in Section 5.1 result in linear systems involving such resistive SPDDD matrices.

5.3 Problem Description

The objective is to solve the linear system

$$Ax = b \tag{5.2}$$

where A is a resistive SPDDD matrix whose associated graph $G(\mathrm{A})$ is a $\sqrt{n} \times \sqrt{n}$ grid graph $G_{\sqrt{n}}$. We further assume that all non-zero off-diagonal entries in A are equal. Following the notation in [25] and [40] we define $G_{\sqrt{n}} = (V, E)$, embedded in plane as follows:

$$\begin{aligned} V &= \{(i,j) \mid 1 \le i \le \sqrt{n}, 1 \le j \le \sqrt{n}\} \\ E &= \{\{(i,j),(i,j+1)\} \mid 1 \le i \le \sqrt{n}, 1 \le j \le \sqrt{n}-1\} \\ &\cup \{\{(i,j),(i+1,j)\} \mid 1 \le i \le \sqrt{n}-1, 1 \le j \le \sqrt{n}\} \end{aligned}$$

Please see Figure 5.1 for a pictorial representation of $G_{\sqrt{n}}$ in which all edge weights are same.

The topology of the grid graph implies that the linear system 5.2 has the following properties.

1. $A \in \Re^{n \times n}$, $b \in \Re^n$, $x \in \Re^n$
2. A is sparse.

Since A is sparse SPDDD matrix, one of the choices to solve the linear system 5.2 is to apply Chebyshev iterative method (see [23]) to the preconditioned system

$$M^{-1}Ax = M^{-1}b \tag{5.3}$$

However, for this iterative method to be fast, M should have the following characteristics which make it a good preconditioner to A.

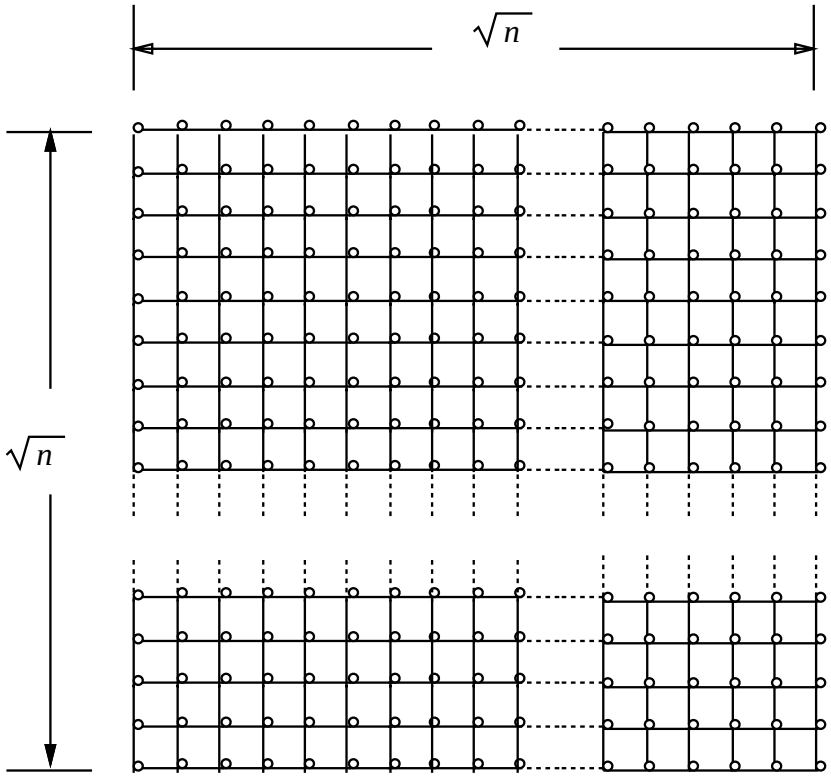


Figure 5.1: Grid Graph

1. M is a good approximation to A. This is quantified by a bound on the condition number of the preconditioned matrix $M^{-1/2}AM^{-1/2}$. A small condition number yields low number of iterations in the iterative method (the number of iterations are also dependent on the stopping criteria used which are usually based on the relative reduction in the two-norm of the residue or $A^{\frac{1}{2}}$ norm of the error).

2. The linear system $Mz = r$ can be solved rapidly after preprocessing of M.

In the following sections we show that it is possible to construct a good preconditioner M for A. Specifically,we show that the following holds for the preconditioner M.

$$\forall y \neq 0,\ 1 \ \leq \ \frac{y^T A y}{y^T M y} \ \leq \ \kappa \tag{5.4}$$

The quantity κ in equation 5.4 is small implying in turn that the condition number of the preconditioned matrix $M^{-1/2}AM^{-1/2}$ is small.

In the next section we describe a sparsification procedure to obtain a preconditioner M to A. Once M is obtained, we show that the problem of solving $Mz = r$ after some preprocessing can be reduced to a problem of solving a smaller system of linear equations whose matrix corresponds to a smaller grid graph.

5.4 Constructing a Good Preconditioner

We construct a good preconditioner M to A by sparsification of the graph $G(\mathrm{A})$. The sparsification procedure is driven by a theorem in [47] which relates two SPDDD matrices through the ratio of their quadratic forms. We state Vaidya's theorem on approximating SPDDD matrices and then go on to show how the grid graph $G_{\sqrt{n}}$ is sparsified to obtain a good preconditioner M.

5.4.1 Approximating SPDDD Matrices

Before restating the main theorem of [47] without proof, let us define a few notations that will help us state the theorem compactly.

e_{ij} In a graph $G(\mathrm{A}) = (V, E)$, the edge connecting vertices $i, j \in V$ is denoted as e_{ij}.

Sparsification Sparsification of a graph $G(\mathrm{A}) = (V, E)$ is another graph $G(\mathrm{M}) = (V, E_s)$ such that $E_s \subset E$ and $G(\mathrm{M})$ has the same vertex weights as $G(\mathrm{A})$. In other words, sparsification

of a graph is another graph which is derived from the original graph by dropping some edges. Let $E_d = E \setminus E_s$ denote the set of dropped edges.

$\Pi(M)$ With each edge $e_{ij} \in E_d$, associate a unique path π_{ij} in $G(\mathrm{M})$ between vertices i and j such that the weight of each edge on π_{ij} is greater than or equal to the weight of the dropped edge e_{ij}. Let $\Pi(M)$ denote the set of these paths. The cardinality of this set is equal to the number of dropped edges.

We define two more quantities p and q which in some sense quantify the sparsification. Given a graph $G(\mathrm{A})$ let graph $G(\mathrm{M})$ be a sparsification of $G(\mathrm{A})$ and $\Pi(M)$ a set of paths as described above. Define p and q as follows.

$$p(\Pi(M)) = \max_{e_{ij} \in E_s} \{\text{Number of paths in } \Pi(M) \text{ passing through the edge } e_{ij}\} \tag{5.5}$$

Intuitively, p is a bound on the number of dropped edges which are accounted for by a retained edge in the process of sparsification. It can also be viewed as a measure of increase in the load on a retained edge due to sparsification.

$$q(\Pi(M)) = \max_{\pi_{ij} \in \Pi(M)} \{\text{Path length of path } \pi_{ij}\} \tag{5.6}$$

Intuitively, q is a bound on the length of an alternate path between the end vertices of a dropped edge in the process of sparsification. This can be thought of as a measure of how much the graph connectivity is weakened due to sparsification.

Now we are ready to state the theorem.

Theorem 5.1 (Vaidya's Matrix Approximation Theorem) *Let A be a resistive SPDDD matrix, M be a sparsification of it, and $\Pi(M)$ be a set of paths with respect to M as defined above. Then the following holds:*

$$\forall y \neq 0,\ 1 \ \leq \ \frac{y^T A y}{y^T M y} \ \leq \ p(\Pi(M))q(\Pi(M)) \tag{5.7}$$

□

In other words, as long as we sparsify A such that load on any edge in M is not high and the connectivity is not weakened too much, then M is a good approximation to A.

We state a lemma that would be useful later.

Lemma 5.2 *Let A and M' be two resistive SPDDD matrices such that*

$$\forall y \neq 0,\ 1 \ \leq \ \frac{y^T A y}{y^T M' y} \ \leq \ \kappa \tag{5.8}$$

where

$$M' = L_M \begin{bmatrix} I & 0 \\ 0 & \tilde{M} \end{bmatrix} L_M^T$$

where $\tilde{M}$ is a resistive SPDDD matrix whose non-zero off-diagonal entries are in the range $[-\beta, -\alpha]$ and $\beta > \alpha > 0$. Let $\hat{M}$ be obtained from $\tilde{M}$ by replacing each non-zero off-diagonal entry with $-\alpha$ and modifying the diagonals such that the row-sums are maintained. Let

$$M = L_M \begin{bmatrix} I & 0 \\ 0 & \hat{M} \end{bmatrix} L_M^T$$

Then

$$\forall y \neq 0,\ 1 \ \leq \ \frac{y^T A y}{y^T M y} \ \leq \ \frac{\beta}{\alpha}\kappa \tag{5.9}$$

Proof:

$$\forall z \neq 0,\ 1 \ \leq \ \frac{z^T \tilde{M} z}{z^T \hat{M} z} \ \leq \ \frac{\beta}{\alpha}$$

which implies

$$\forall y \neq 0,\ 1 \ \leq \ \frac{y^T M' y}{y^T M y} \ \leq \ \frac{\beta}{\alpha}$$

from which Equation 5.9 follows. □

5.4.2 Approximations of Grid Graphs

In this section, we give a sparsification procedure for grid graph $G_{\sqrt{n}}$ which is the associated graph of a resistive SPDDD matrix A whose non-zero off-diagonal entries are equal. This sparsification M can be used as a preconditioner to A. Let m be a parameter that will be chosen later.

Sparsification of a Block of Size $s \times t$

We give a sparsification procedure for a block of size $s \times t$ of a grid graph denoted by $B_{s\times t}$ where $s, t \in [\sqrt{m}, 2\sqrt{m})$. This procedure will be used later to sparsify A. Please refer to Figure 5.2.

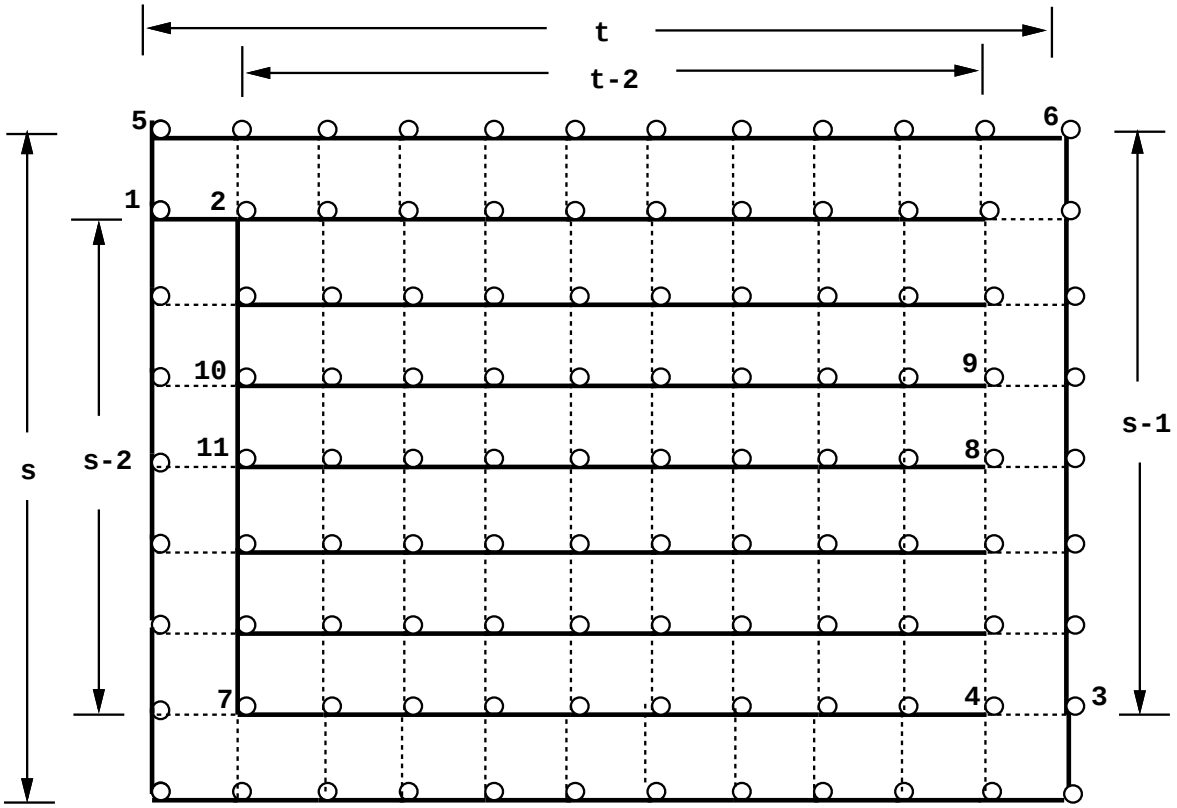


Figure 5.2: Sparsification of Block $B_{s\times t}$

Block $B_{s\times t}$ consists of an inner block $B_{s-2\times t-2}$ whose boundary is connected to the boundary of $B_{s\times t}$ with a total of $2(s+t)-8$ edges. Let us call these the type 1 edges. We disconnect the boundary of $B_{s\times t}$ from the inner block $B_{s-2\times t-2}$ by dropping all but one of these edges. The edge that is retained to maintain connectivity is the edge between vertices 1 and 2. We sparsify the inner block $B_{s-2\times t-2}$ by constructing a spanning tree of diameter $2t+s-9 \leq 6\sqrt{m}$. Let us call the non-tree edges which are dropped from the inner block $B_{s-2\times t-2}$ the type 2 edges. To get bounds on the quantities p and q of Theorem 5.1 for this sparsification of a block $B_{s\times t}$ we need to assign unique paths for each dropped edge.

For every dropped type 1 edge $\{i, j\}$, we note that one of the vertices, say i, is on the boundary of block $B_{s\times t}$ and the other vertex j is on the boundary of the inner block $B_{s-2\times t-2}$. The unique path we assign to type 1 edge $\{i, j\}$ consists of two segments, one segment is the shortest path from vertex i to vertex 1 along the boundary of block $B_{s\times t}$ and the other segment is the unique path from 1 to vertex j along the spanning tree on the inner block $B_{s-2\times t-2}$. For example, the unique path assigned to the dropped edge $\{3, 4\}$ is the path through vertices (3,6,5,1,2,7,4).

For every dropped type 2 edge $\{i, j\}$, the unique path assigned is the unique tree path between i and j in the spanning tree on the inner block $B_{s-2\times t-2}$. For example, the unique path assigned to the dropped edge $\{8, 9\}$ is the path through vertices (8,11,10,9).

It is clear from Figure 5.2 that only the paths assigned to type 1 dropped edges go through edge $\{1, 2\}$. Furthermore, the maximum number of paths go through the edge $\{1, 2\}$. Thus from Equation 5.5, we can put an upper bound on p of $2(s + t) - 9 \leq 8\sqrt{m}$. The maximum length path is the unique path assigned to the dropped edge $\{3, 4\}$. Thus from Equation 5.6, we can put an upper bound on q of $2(s + t) - 7 \leq 8\sqrt{m}$.

Sparsification of Grid Graph $G_{\sqrt{n}}$

The following sparsification procedure is applied to $G_{\sqrt{n}}$ to obtain a good approximation M' to A. The procedure is described in terms of the sparsification of a block $B_{s\times t}$ as described previously.

1. Divide the grid graph $G_{\sqrt{n}}$ into $\frac{n}{m}$ blocks each of size $\sqrt{m} \times \sqrt{m}$. If $\sqrt{m}$ does not divide $\sqrt{n}$ properly, then this will lead to a set of partial blocks at the bottom and right edges of $G_{\sqrt{n}}$. We merge these partial blocks into the last complete blocks either on top or to the left of them. This will result in all blocks of type $B_{s\times t}$ where $s, t \in [\sqrt{m}, 2\sqrt{m})$. Please refer to Figure 5.3.

2. Apply the block sparsification procedure described above to each of these blocks. The unique path assignment is same as in the block sparsification procedure. The sparsification of $G_{\sqrt{n}}$ would look like Figure 5.4.

   We can now compute p and q as defined in Equations 5.5 and 5.6. For computation of p, note that the number of unique paths going through any boundary edge that is shared between two blocks is bounded from above by the total of dropped type 1 edges from each of the blocks which in turn is bounded from above by $16\sqrt{m}$. A little more reflection reveals that

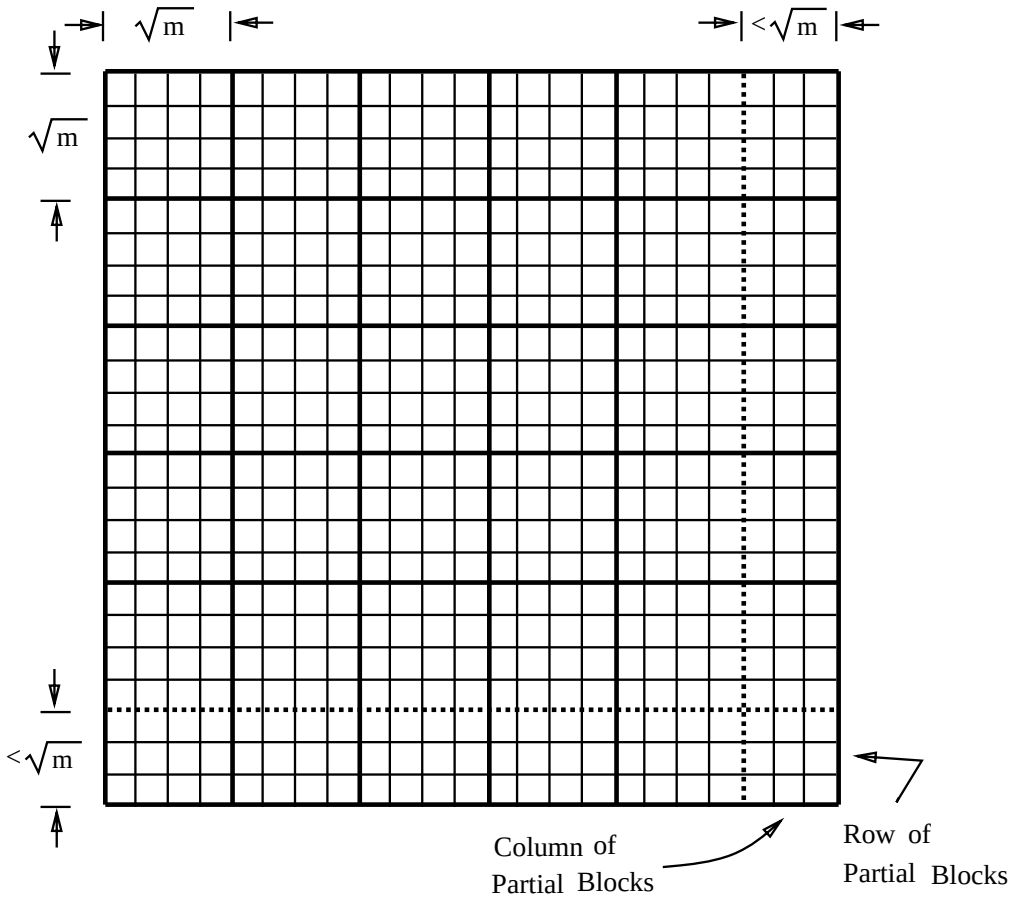


Figure 5.3: Division of the Grid Graph into Blocks

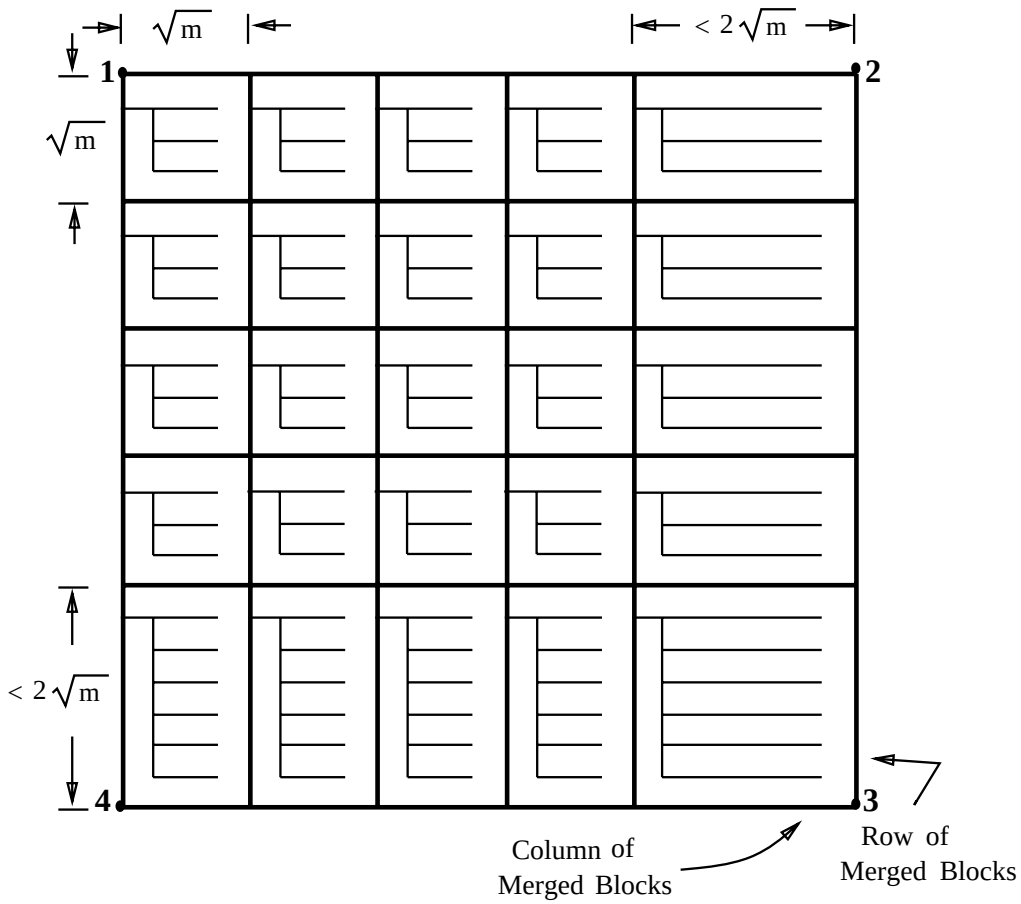


Figure 5.4: Block Sparsification Applied to the Grid Graph

the bound is still $8\sqrt{m}$. The upper bound on q is the same as before, i.e. $8\sqrt{m}$. This results in $pq \leq 64m$.

Now we can apply Theorem 5.1 as follows. If the grid graph $G_{\sqrt{n}}$ is obtained from a resistive SPDDD matrix A with all of its non-zero off-diagonals being equal, then the sparsified graph of Figure 5.4 gives us the approximation M' to A whose approximation quality is governed by

$$\forall y \neq 0,\ 1 \ \leq \ \frac{y^T A y}{y^T M' y} \ \leq \ pq \ \leq \ C_1 m \tag{5.10}$$

where $C_1 = 64$. This means that the condition number of the matrix $M'^{-1/2} A M'^{-1/2}$ is upper bounded by $C_1 m$ for this sparsification procedure.

Compaction of the Sparsified Graph

As we shall see later, we need to solve a linear system involving the preconditioner in every iteration of the Chebyshev iterative procedure. The solve will be facilitated if we can simplify the preconditioner M' to another matrix M such that the quality of approximation is not really affected.

To simplify the structure of M', we apply an operation which we call Compaction on this graph which consists of two steps:

1. Iteratively eliminate all degree one vertices.

2. Iteratively eliminate degree two vertices excepting the corner vertices, i.e. vertices 1,2,3 and 4.

Compaction can be considered as the process of applying a partial Gaussian elimination on the matrix M' in which the order of elimination is given by the vertex ordering produced by each of the steps of Compaction. To simplify matters we assume that the vertices are already numbered in the order that is dictated by Compaction.

Compaction on M' converts the graph of Figure 5.4 to the graph of Figure 5.5 and also produces a partial factor L_M and its transpose L_M^T.

In algebraic terms, this can be expressed as a factoring of M' into three matrices as follows:

$$M' = L_M \begin{bmatrix} I & 0 \\ 0 & \tilde{M} \end{bmatrix} L_M^T \tag{5.11}$$

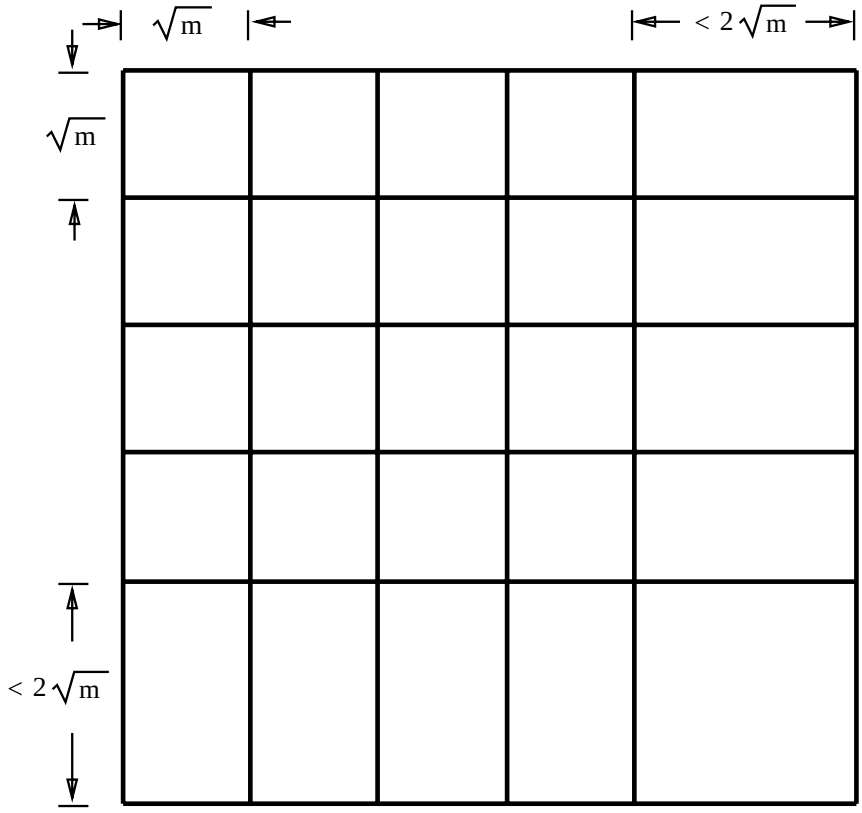


Figure 5.5: Grid Graph After Sparsification and Compaction

where $\tilde{M}$ is a resistive SPDDD matrix whose non-zero off-diagonals are not equal and its associated graph is the graph of Figure 5.5.

To draw this connection between $\tilde{M}$ and Figure 5.5 let us examine each step of Compaction in detail.

The first step of Compaction eliminates all the vertices of the spanning trees which are interior to each of the $\frac{n}{m}$ blocks $B_{s\times t}$ of Figure 5.4. This does not cause any fill-in.

The second step of Compaction corresponds to the shrinking of series conductances. When we eliminate a vertex i of degree two whose two neighbors are j and k, then j and k get connected resulting in O(1) fill-in. Consider the state of the graph of Figure 5.4 at the end of the first step of Compaction. We are left with $\frac{n}{m}$ blocks; each $s \times t$ block has an empty interior and is bounded by $s-1$ series conductances on the left and right, and $t-1$ series conductances on the top and bottom. Let the value of the conductances in M' be $\frac{1}{\gamma}$. When we apply the second step of Compaction, the series conductances on the left (and right) and top (and bottom) shrink to single conductances of values $\frac{1}{(s-1)\gamma}$ and $\frac{1}{(t-1)\gamma}$ respectively. Thus what we obtain is a grid graph $G_{\lfloor\sqrt{\frac{n}{m}}\rfloor}$ of Figure 5.5 whose associated matrix is $\tilde{M}$.

The argument about fill-in given above implies that L_M in Equation 5.11 has O(n) non-zeros. The matrix $\tilde{M} \in \Re^{\frac{n}{m}\times\frac{n}{m}}$ is a resistive SPDDD matrix whose non-zero off-diagonal entries are in the range $[-\frac{1}{\sqrt{m}\gamma}, -\frac{1}{2\sqrt{m}\gamma})$.

Define

$$M = L_M \begin{bmatrix} I & 0 \\ 0 & \hat{M} \end{bmatrix} L_M^T \tag{5.12}$$

where $\hat{M}$ is a resistive SPDDD matrix which has the same zero/non-zero pattern as $\tilde{M}$ but all of its non-zero off-diagonals are equal to $-\frac{1}{2\sqrt{m}\gamma}$ and each of its diagonal is modified such that the row-sums are maintained to be equal to those of $\tilde{M}$.

By applying Lemma 5.2 to equations 5.10, 5.11, and 5.12, we get the following inequality which relates the quadratic forms of A and M.

$$\forall y \neq 0,\ 1 \ \leq\ \frac{y^T A y}{y^T M y} \ \leq\ 2C_1 m \tag{5.13}$$

We close this section by giving the main result which is

Theorem 5.3 *Let $A \in \Re^{n \times n}$ be a resistive SPDDD matrix with all of its non-zero off-diagonals being equal whose associated graph is a grid graph $G_{\sqrt{n}}$ and m be a parameter to be chosen. Then we can construct an approximation matrix*

$$M = L_M \begin{bmatrix} I & 0 \\ 0 & \hat{M} \end{bmatrix} L_M^T$$

where $\hat{M} \in \Re^{\frac{n}{m} \times \frac{n}{m}}$ is a resistive SPDDD matrix with all of its non-zero off-diagonals being equal whose associated graph is a grid graph $G_{\lfloor\sqrt{\frac{n}{m}}\rfloor}$. Moreover, the quality of the approximation is given by

$$\forall y \neq 0,\ 1 \ \leq \ \frac{y^T A y}{y^T M y} \ \leq \ C_2 m \tag{5.14}$$

5.5 Solution Methodology

The matrix A in the linear system 5.2 is sparse. As a result iterative methods are better suited than direct methods for the solution of this linear system. We choose to analyze the quality of the preconditioner for the Chebyshev iterative method. For a good description of Chebyshev iterative method see [22], [34], [21] or [23].

In this section we give an overview of a general solution method which can be specialized into two distinct algorithms - a two-level algorithm which uses Chebyshev iterative method along with Nested Dissection (see [19],[40]) for factoring the preconditioner, and a recursive algorithm which employs the Chebyshev iterative method recursively. The first algorithm solves linear system 5.2 in $O(n^{1+\frac{1}{8}})$ operations, whereas the recursive algorithm solves the same linear system in $O(n)$ operations, both algorithms solving the linear system to a fixed degree of accuracy. We also give a description of the Chebyshev iterative method as described in [21] and a formula for the number of operations for the general procedure.

5.5.1 Overview of the Solution Method

The major steps in our solution method are as follows.

1. Construct a preconditioner M to A by applying the sparsification procedure as described in Section 5.4.2. The quadratic forms of A and M are related by the inequality 5.14. This step takes $O(n)$ operations.

2. Solve the linear system $Ax = b$ using the Chebyshev iterative method with M as the preconditioning matrix. As we shall see later the following steps need to be carried out in each Chebyshev iteration.

   (a) Multiply a vector $x \in \Re^n$ with A. This takes number of operations proportional to the number of non-zeros in A, namely $O(n)$ operations.

   (b) Solve the linear system $Mz = r$ for z where $z, r \in \Re^n$. But M has the special structure as described in Equation 5.12. This means that the solution of $Mz = r$ can be broken up into three components.

      i. Forward substitute with L_M which takes $O(n)$ operations.

      ii. Solve for $\hat{z}$ the smaller linear system

$$\hat{M}\hat{z} = \hat{r} \tag{5.15}$$

      where $\hat{M} \in \Re^{\frac{n}{m} \times \frac{n}{m}}$ with the associated graph which is $G_{\lfloor\sqrt{\frac{n}{m}}\rfloor}$. Also $\hat{z}, \hat{r} \in \Re^{\frac{n}{m}}$. Let , denote the number of operations to solve this smaller linear system 5.15 assuming that $\hat{M}$ has been possibly pre-processed for convenience. Let , $_p$ denote the one-time preprocessing cost for preprocessing $\hat{M}$.

      iii. Back solve with L_M^T which also takes $O(n)$ operations.

It is clear that the total number of operations to solve the linear system 5.2 depends on the number of iterations, and , . In the Chebyshev method, the number of iterations is dependent on the condition number of $M^{-1/2}AM^{-1/2}$(which is bounded by $\kappa = C_2 m$) and the accuracy ξ to which the linear system needs to be solved. Specifically the number of iterations needed to solve the linear system 5.2 to an accuracy ξ is $C_3\sqrt{m}\log\frac{1}{\xi}$ where C_3 is a large enough constant. This is so because the Chebyshev iterative method has a convergence rate such that in a number of iterations proportional to the square root of the condition number of the underlying matrix, the error is cut down by a constant factor, as we shall see later.

Thus the total number of operations to solve linear system 5.2 to an accuracy ξ, denoted by $T(n, \xi)$, is given by

$$T(n,\xi) \;=\; C_3\sqrt{m}\,(\,,\; + O(n))\;\log(1/\xi) \;+\; O(n) \;+\; ,_p \tag{5.16}$$

In the above equation the first term dominates the second term giving

$$T(n,\xi) \;=\; C_3\sqrt{m}\,(,\; + \mathrm{O}(n))\;\log(1/\xi)\;+\;,_{\,p} \tag{5.17}$$

5.5.2 Exact and Inexact Chebyshev Iterative Methods

First we give the exact Chebyshev iterative method and then quote a theorem from [21] without proof which characterizes the convergence of inexact Chebyshev iterative method.

5.5.3 Exact Chebyshev Iterative Method

Given the linear system 5.2, we can solve it using the following iterative scheme which is known as the Chebyshev iterative method.

$$x_1 = \alpha z_0 \tag{5.18}$$

$$x_{k+1} = \omega_{k+1}(\alpha z_k + x_k) + (1-\omega_{k+1})x_{k-1},\; k = 1,2,\ldots \tag{5.19}$$

where

$$Mz_k = r_k \tag{5.20}$$

$$r_k = b - Ax_k \tag{5.21}$$

$$\alpha = \frac{2}{u+l} \tag{5.22}$$

$$\mu = \frac{u+l}{u-l} \tag{5.23}$$

$$\omega_{k+1} = 2\mu\frac{c_k}{c_{k+1}} \tag{5.24}$$

$$c_{k+1} = 2\mu c_k - c_{k-1},\; k = 1,2,\ldots \tag{5.25}$$

$$c_0 = 1, c_1 = \mu \tag{5.26}$$

where l and u are lower and upper bounds on the eigenvalues of $M^{-1}A$ (which has the same eigenvalues as $M^{-1/2}AM^{-1/2}$). The parameter α is chosen such that the spectral radius of the iteration matrix $(I - \alpha M^{-1}A)$ is less than 1. The quantity c_k is the value of the k^{th} Chebyshev polynomial evaluated at μ. Using the inequality 5.14, we can replace l and u with 1 and $\kappa(= C_2 m)$

respectively which results in

$$\kappa = C_2 m \tag{5.27}$$

$$\alpha = \frac{2}{\kappa + 1} \tag{5.28}$$

$$\mu = \frac{\kappa + 1}{\kappa - 1} \tag{5.29}$$

From Equations 5.21 and 5.20 we see that in each iteration, we need to multiply a vector x_k with A and also solve a linear system involving the preconditioner M. Note that the linear system $Mz_k = r_k$ is solved as described in Section 5.5.1 and it involves the solution of a smaller linear system $\hat{M}\hat{z}_k = \hat{r}_k$ where $\hat{r_k}$ is obtained by projecting $L_M^{-1} r_k$ onto the last $\frac{n}{m}$ variables.

5.5.4 Inexact Chebyshev Iterative Method

If we assume that the linear system of Equation 5.20 is solved inexactly, then the convergence of the Chebyshev method is different from the exact version. The inexact version of the method is obtained by replacing Equation 5.20 with

$$Mz_k = r_k + q_k \tag{5.30}$$

$$\frac{\parallel M^{-\frac{1}{2}} q_k \parallel}{\parallel M^{-\frac{1}{2}} r_k \parallel} \leq \delta, \ \delta \in (0,1) \tag{5.31}$$

which can also be written as

$$\frac{\parallel M^{\frac{1}{2}} (z_k^* - z_k \parallel}{\parallel M^{\frac{1}{2}} z_k \parallel} \leq \delta, \ \delta \in (0,1) \tag{5.32}$$

where z_k^* is the true solution of the linear system 5.20. In other words, δ is the relative reduction in the matrix norm of the error.

We state a theorem which is a direct consequence of *Theorem 2* from [21] when A and M are SPDDD matrices. This theorem gives us a bound on the convergence rate of the inexact Chebyshev method.

Theorem 5.4 *Assume M and A are SPDDD matrices. The error in the inexact Chebyshev method is bounded by*

$$\parallel M^{\frac{1}{2}} e_k \parallel \leq B_E B_I \parallel M^{\frac{1}{2}} e_0 \parallel \tag{5.33}$$

where

$$B_E = \frac{1}{\cosh k \cosh^{-1} \mu} \tag{5.34}$$

$$B_I = \cosh k \cosh^{-1}(1+\varepsilon\mu) + \varepsilon\mu \frac{\sinh k \cosh^{-1}(1+\varepsilon\mu)}{\sinh \cosh^{-1}(1+\varepsilon\mu)} \tag{5.35}$$

$$\varepsilon = \delta\alpha \parallel M^{-1/2} A M^{-1/2} \parallel \tag{5.36}$$

where α, μ, and δ are given by 5.28,5.29 and 5.31 (or 5.32) respectively. $e_k = x^* - x_k$ and $e_0 = x^*$ where x^* is the true solution to the linear system 5.2. □

Remark We briefly outline how Theorem 5.4 relates to *Theorem 2* of [21]. The term ρ in *Theorem 2* of [21] is 1 because $-1 \leq \mu\sigma_1 \leq 1$ which is a direct consequence of A and M being SPDDD matrices. By slightly redefining δ as given by Equation 5.31, and the extra information that A and M are Symmetric Positive Definite, a tighter relation between ε and δ as defined by Equation 5.36 can be obtained. We choose to measure the relative error in the solutions of the linear systems $Mz_k = r_k$ and $Ax = b$ using the same type of matrix norm which allows for a simplification of the analysis.

Corollary 5.5 *In addition to the assumptions of Theorem 5.4, if we also assume that $\varepsilon\mu \leq \frac{1}{100\kappa}$, then*

$$B_E B_I \leq 6(1 - \frac{1}{\sqrt{\kappa}})^k \tag{5.37}$$

and

$$\frac{\parallel A^{\frac{1}{2}} e_k \parallel}{\parallel A^{\frac{1}{2}} x^* \parallel} \leq 6\sqrt{\kappa}(1 - \frac{1}{\sqrt{\kappa}})^k \tag{5.38}$$

Proof: An upper bound on B_E can be obtained from the definition of $\cosh k \cosh^{-1} \mu$ which is given by

$$B_E \leq 2(\frac{1 - \frac{1}{\sqrt{\kappa}}}{1 + \frac{1}{\sqrt{\kappa}}})^k$$

We get the upper bound on B_I as follows.

$$B_I = \cosh k \cosh^{-1}(1+\varepsilon\mu)(1 + \frac{\varepsilon\mu}{\sinh \cosh^{-1}(1+\varepsilon\mu)} \tanh k \cosh^{-1}(1+\varepsilon\mu))$$

We know that $-1 \leq \tanh k \cosh^{-1}(1+\varepsilon\mu) \leq 1$. Further we can, with a little manipulation, show that

$$\frac{\varepsilon\mu}{\sinh \cosh^{-1}(1+\varepsilon\mu)} \leq 2\sqrt{\varepsilon\mu}$$

Thus

$$B_I \leq (1+2\sqrt{\varepsilon\mu}) \cosh k \cosh^{-1}(1+\varepsilon\mu)$$

From the definition of $\cosh k \cosh^{-1}(1+\varepsilon\mu)$ we get an upper bound on B_I as follows.

$$B_I \leq (1+2\sqrt{\varepsilon\mu})(\frac{1+\sqrt{\varepsilon\mu}}{1-\sqrt{\varepsilon\mu}})^k$$

and hence

$$B_E B_I \leq 2(1+2\sqrt{\varepsilon\mu})(\frac{1-\frac{1}{\sqrt{\kappa}}}{1+\frac{1}{\sqrt{\kappa}}} \ \frac{1+\sqrt{\varepsilon\mu}}{1-\sqrt{\varepsilon\mu}})^k$$

Letting $\varepsilon\mu \leq \frac{1}{100\kappa}$ and $\kappa \geq 10$ gives us the desired bound on $B_E B_I$ which is

$$B_E B_I \leq 6(1-\frac{1}{\sqrt{\kappa}})^k$$

The inequality 5.38 can be derived as follows.

$$\frac{\| A^{\frac{1}{2}} e_k \|}{\| A^{\frac{1}{2}} e_0 \|} = \sqrt{\frac{e_k^T A e_k}{e_k^T M e_k} \ \frac{e_k^T M e_k}{e_0^T M e_0} \ \frac{e_0^T M e_0}{e_0^T A e_0}}$$

which when combined with inequality 5.14 results in

$$\frac{\| A^{\frac{1}{2}} e_k \|}{\| A^{\frac{1}{2}} e_0 \|} \leq \sqrt{\kappa} \frac{\| M^{\frac{1}{2}} e_k \|}{\| M^{\frac{1}{2}} e_0 \|}$$

From inequalities 5.37 and 5.33, we get the required result, i.e.

$$\frac{\| A^{\frac{1}{2}} e_k \|}{\| A^{\frac{1}{2}} x^* \|} \leq 6\sqrt{\kappa}(1-\frac{1}{\sqrt{\kappa}})^k$$

□

Corollary 5.6 *Assume M and A are SPDDD matrices. The error in the exact Chebyshev method is bounded by*

$$\frac{\| M^{\frac{1}{2}} e_k \|}{\| M^{\frac{1}{2}} e_0 \|} \leq B_E \tag{5.39}$$

where

$$B_E = \frac{1}{\cosh k \cosh^{-1} \mu} \leq 2(1 - \frac{2}{\sqrt{\kappa}+1})^k \tag{5.40}$$

where μ is given by Equation 5.29. $e_k = x^* - x_k$ and $e_0 = x^*$ where x^* is the true solution to the linear system 5.2. □

5.6 Two Level Algorithm

This algorithm uses the exact Chebyshev method. The steps in the algorithm are

1. Given the linear system 5.2 where A corresponds to the grid graph $G_{\sqrt{n}}$, derive the preconditioner M as described in Section 5.4.2.

2. Apply the Chebyshev method as described above. We use the exact Chebyshev method in that the matrix in linear system 5.15 is factored exactly using the strategy of Nested Dissection(see [40]). Since we are applying Nested Dissection on a grid graph of size $\frac{n}{m} \times \frac{n}{m}$, the one-time factoring cost is , $_p$ = O($(\frac{n}{m})^{\frac{3}{2}}$). Since the fill-in is O($\frac{n}{m} \log (\frac{n}{m})$), the factor of $\hat{M}$ has O($\frac{n}{m} \log (\frac{n}{m})$) non-zeros and the cost of solving the linear system 5.15 in every iteration is , = O($\frac{n}{m} \log (\frac{n}{m})$).

We need to compute the number of iterations to reduce the relative error such that $\frac{\|M^{\frac{1}{2}} e_k\|}{\|M^{\frac{1}{2}} e_0\|} \leq \xi$. From inequality 5.39, we can conclude that the number of iterations k should satisfy

$$2(1 - \frac{2}{\sqrt{\kappa}+1})^k \leq \xi$$

which leads us to

$$k = \mathrm{O}(\sqrt{\kappa} \log \tfrac{1}{\xi})$$

from which we conclude that to achieve an accuracy of ξ, we need $C_3\sqrt{m}\log\frac{1}{\xi}$ iterations. The number of operations in the two-level algorithm is given by

$$T(n,\xi) \;=\; C_3\sqrt{m}\;(C_4(\frac{n}{m})\;\log(\frac{n}{m}) + \mathrm{O}(n))\;\log\frac{1}{\xi} \;+\; C_5(\frac{n}{m})^{\frac{3}{2}} \tag{5.41}$$

In the above equation, $\mathrm{O}(n)$ will dominate $C_4(\frac{n}{m})\log(\frac{n}{m})$ because we will have to choose an m such that $m = \Omega(\log n)$. The rate of growth of $T(n,\xi)$ is minimized by picking an m satisfying

$$\log(1/\xi)\sqrt{m}n = (\frac{n}{m})^{\frac{3}{2}}$$

which gives us $m = \frac{n^{\frac{1}{4}}}{(\log(1/\xi))^{\frac{1}{2}}}$. Thus the number of operations in the two-level algorithm is given by

$$T(n,\xi) = \mathrm{O}(n^{1+\frac{1}{8}}(\log(1/\xi))^{\frac{3}{4}})$$

If Strassen's fast matrix multiplication is used in the Nested Dissection (see [40]), then the above bound can be refined to $\mathrm{O}(n^{1.106}\log^{0.738}(1/\xi))$.

5.7 Recursive Algorithm

We observe that the smaller linear system 5.15 is same as the original linear system 5.2 in that each of A and $\hat{M}$ is a resistive SPDDD matrix with all of its non-zero off-diagonals being equal and their associated graphs are grid graphs $G_{\sqrt{n}}$ and $G_{\lfloor\sqrt{\frac{n}{m}}\rfloor}$ respectively. Hence we can apply the Chebyshev iterative method recursively to solve the smaller linear system 5.15 to a fixed accuracy in each iteration of the Chebyshev iterative method. Since the smaller linear system 5.15 is solved inexactly using Chebyshev iterative method, we have to use Theorem 5.4 and Corollary 5.5 to get a bound on the number of iterations.

5.7.1 Analysis of the Recursive Algorithm

We give a theorem which will be useful in the analysis.

Theorem 5.7 *Let $\delta \leq \frac{\kappa-1}{200\kappa^2}$ and $\kappa = C_2 m \geq 10$. Also suppose that we have a solution to the smaller linear system 5.15*

$$\hat{M}\hat{z_k} = \hat{r_k}, \quad \hat{M} \in \Re^{\frac{n}{m}\times\frac{n}{m}}$$

such that

$$\frac{\| \hat{M}^{\frac{1}{2}} \hat{e_k} \|}{\| \hat{M}^{\frac{1}{2}} \hat{z_k^*} \|} \leq \delta \tag{5.42}$$

where $\hat{z_k^*}$ *is the true solution and* $\hat{e_k}$ *the error in the computed solution. Then we can get a convergence rate for the linear system 5.2 which is given by*

$$\frac{\| A^{\frac{1}{2}} e_k \|}{\| A^{\frac{1}{2}} x^* \|} \leq 6\sqrt{\kappa}(1 - \frac{1}{\sqrt{\kappa}})^k \tag{5.43}$$

Proof: Let z_k^* be the true solution to Equation 5.20. Since M is related to $\hat{M}$ through Equation 5.12 and solves with L_M and L_M^T are done exactly, from inequality 5.42 it can be easily shown that

$$\frac{\| M^{\frac{1}{2}} (z_k - z_k^*) \|}{\| M^{\frac{1}{2}} z_k^* \|} \leq \delta \tag{5.44}$$

Note that $\| M^{-1/2} A M^{-1/2} \| \leq \kappa$. Also, since $\delta \leq \frac{\kappa - 1}{200\kappa^2}$, from Equations 5.28, 5.29, and 5.36 it follows that $\varepsilon\mu \leq \frac{1}{100\kappa}$. The required result then follows from Equation 5.44, Theorem 5.4, and Corollary 5.5. □

Using the above theorem, we can compute the number of iterations required to reduce the relative $A^{\frac{1}{2}}$ norm of the error by δ. From inequality 5.38 we get that, the number of iterations k must satisfy

$$6\sqrt{\kappa}(1 - \frac{1}{\sqrt{\kappa}})^k \leq \delta$$

Thus

$$(1 - \frac{1}{\sqrt{\kappa}})^k \leq \frac{1}{1200\kappa\sqrt{\kappa}}$$

$$k \geq C_6 \sqrt{m} \log m \tag{5.45}$$

The above inequality says that in a number of iterations proportional to the square root of the condition number of the preconditioned system, the error is reduced by a constant factor, even when the inner solve is done inexactly, provided the inner system is solved to sufficient accuracy, i.e. solved to an accuracy of at least δ. From this inequality we can write the recurrence for the number of operations in the recursive algorithm. The recurrence is similar to Equation 5.17, the difference being , $_p$ vanishes because there is no preprocessing to be performed to solve the smaller linear system. ξ is nothing but δ which is only a function of the condition number of the

preconditioned system. , is the number of operations to solve the smaller linear system 5.15 which is solved recursively.

Let $T(n)$ be the number of operations to solve the linear system $Ax = b$ using the recursive algorithm just described where $A \in \Re^{n \times n}$, $x, b \in \Re^n$ and A is a sparse SPDDD matrix with all of its non-zero off-diagonals being equal whose associated graph is a grid graph $G_{\sqrt{n}}$. Then $T(n)$ is given by the following recurrence relation.

$$T(n) \leq C_6\sqrt{m}\ \log m\ (C_7 n\ +\ T(\frac{n}{m})) \tag{5.46}$$

Let m be such that

$$m \geq \max\ \{(C_6C_7 + C_6)^4,\ 10\} \tag{5.47}$$

We want to show that

$$T(n) \leq nm \tag{5.48}$$

Assume, by way of induction, that

$$T(\frac{n}{m}) \leq n \tag{5.49}$$

Substituting inequality 5.49 into inequality 5.46 and comparing with the quantity on the right hand side of inequality 5.48, we need

$$T(n) \leq C_6\sqrt{m}\log m(C_7 n + n) \leq nm$$

But we know that $\log m \leq m^{\frac{1}{4}}$ from inequality 5.47. Hence we need to show

$$T(n)\ \leq\ (C_6C_7 + C_6)\ nm^{\frac{3}{4}}\ \leq\ nm$$

Again from from inequality 5.47 we know that $m^{\frac{1}{4}} \geq (C_6C_7 + C_6)$ from which we conclude that

$$T(n) \leq nm$$

Thus we conclude that $T(n) = \mathrm{O}(n)$ operations are required to solve the linear system 5.2. Note that, to achieve this bound, we need to choose the parameter m to be large enough and is given by inequality 5.47.

Note that it is possible to tighten the constants in the analysis. C_6 can be made small if n is a power of two. Tightening the constants in the error analysis too would make the constant C_6 smaller. Especially, the bound on δ can be tightened further. We predict that, in practice, an accuracy of δ (or smaller) will be achieved in far fewer iterations than $C_6\sqrt{m}\log m$.

5.8 Other Recursive Topologies

In this section we point out some other topologies for which the above analysis would work without modifications. For the analysis to apply, the linear system should satisfy the following criteria.

1. The matrix must be a resistive SPDDD matrix with all of its non-zero off-diagonals are equal or in the range $[-\beta, -\alpha]$ with $\beta > \alpha > 0$. In the latter case, we replace A of linear system 5.2 with another matrix A' which has all of its non-zero off-diagonals equal to $-\alpha$ and the row-sums are maintained to be those of A. We use A' as the preconditioner to A and apply the recursive algorithm for solving the linear system $A'z_k = r_k$ which arises in each iteration of the Chebyshev iterative method. By an application of a slightly specialized version of Lemma 5.2, we see that the asymptotic behavior will not change.

2. It is possible to sparsify the graph associated with A such that the product $p(\Pi(M))q(\Pi(M))$ is bounded from above. This will put a bound on the number of iterations, which in turn bounds the number of operation.

3. After sparsification, compaction, and equalization of non-zero off-diagonals (all of these operations have been defined previously), the resulting graph should belong to the same class as the graph we started with. This will allow us to apply the solution method recursively for a smaller linear system whose matrix belongs to the same class as of the matrix of the linear system 5.2.

Now we will give some examples for which we outline the sparsification procedure and give the upper bound on the product $p(\Pi(M))q(\Pi(M))$.

Three Dimensional Grid Graphs

The definition of a two-dimensional grid graph can be naturally extended to a three-dimensional grid graph $G_{\sqrt[3]{n}} = (V, E)$ where V and E are defined as follows.

$$\begin{aligned} V &= \{(i,j,k) \mid 1 \le i \le \sqrt[3]{n}, 1 \le j \le \sqrt[3]{n}, 1 \le k \le \sqrt[3]{n}\} \\ E &= \{\{(i,j,k),(i+1,j,k)\} \mid 1 \le i \le \sqrt[3]{n}-1, 1 \le j \le \sqrt[3]{n}, 1 \le k \le \sqrt[3]{n}\} \\ &\cup \{\{(i,j,k),(i,j+1,k)\} \mid 1 \le i \le \sqrt[3]{n}, 1 \le j \le \sqrt[3]{n}-1, 1 \le k \le \sqrt[3]{n}\} \\ &\cup \{\{(i,j,k),(i,j,k+1)\} \mid 1 \le i \le \sqrt[3]{n}, 1 \le j \le \sqrt[3]{n}, 1 \le k \le \sqrt[3]{n}-1\} \end{aligned}$$

Let us describe the sparsification procedure for a size $s \times t \times u$ block $B_{s \times t \times u}$where s, t, u are in the range $[\sqrt[3]{m}, 2\sqrt[3]{m})$. The vertices on the boundaries of this block are connected to a smaller block $B_{s-2 \times t-2 \times u-2}$ which is "contained" in $B_{s \times t \times u}$. Again we call these the type 1 edges. We drop all but one of the type 1 edges. We also construct an appropriate spanning tree of diameter $O(\sqrt[3]{m})$ on the inner block $B_{s-2 \times t-2 \times u-2}$. The edges which are non-tree edges are dropped. Let us call the non-tree edges type 2 edges. By a similar argument which was applied to the two-dimensional grid graph, we see that $p(\Pi(M))$ is bounded from above by the number of dropped type 1 edges which are $O(m^{\frac{2}{3}})$. The upper bound on $q(\Pi(M))$ is obtained from the upper bound on the unique paths assigned to dropped type 1 edges. This upper bound is $O(\sqrt[3]{m})$. Thus the quality of approximation M to A is dictated by the inequality 5.14 with a different constant C_2. Note that after sparsification, compaction, and equalizing the non-zero off-diagonals, the resulting graph is another three-dimensional grid graph of smaller size. The rest of the analysis is same as for two-dimensional grid graphs.

It is easy to see that the analysis carries through to grid graphs of arbitrary dimension d. The quantity $p(\Pi(M))$ is upper bounded by $O(m^{\frac{d-1}{d}})$ and $q(\Pi(M))$ is upper bounded by $O(m^{\frac{1}{d}})$ which results in an approximation M to A whose quality of approximation is dictated by the inequality 5.14 with the constant C_2 dependent on dimension d.

Recursive Triangular Graph in Two Dimensions

We define the recursive triangular graph to be a large triangle that is recursively subdivided into smaller triangles. This graph is described pictorially in Figure 5.6.

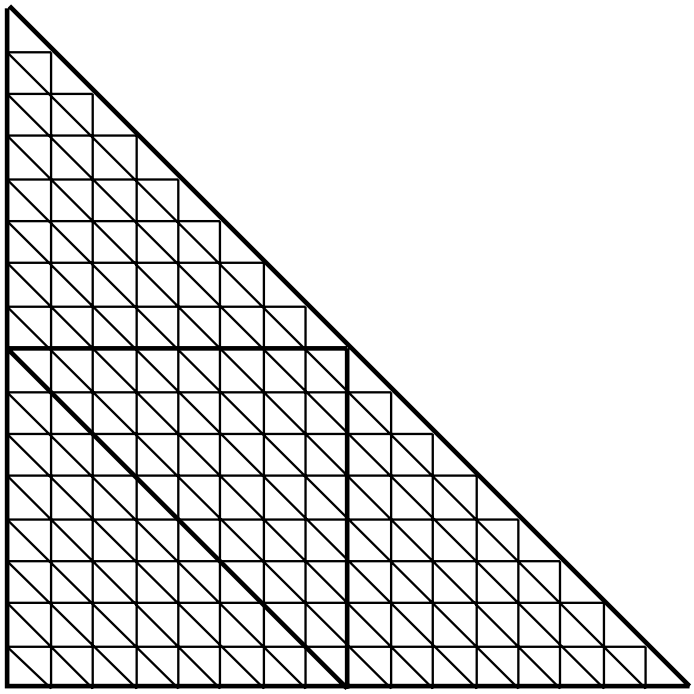


Figure 5.6: Recursive Triangular Graph

The recursive triangular graph is half of a grid graph embedded in a plane with extra edges to completely triangulate it. As a first step in the sparsification procedure, this graph is divided into coarser triangular blocks of smaller size. In the second step of the sparsification procedure, an appropriate spanning tree is found on the inner triangular block of each coarse triangular block. Please refer to Figure 5.7 for details of the sparsification procedure.

The same graph after compaction looks like Figure 5.8.

From these figures, it is clear that the product $p(\Pi(M))q(\Pi(M))$ is bounded from above by a constant. Note that sparsification, compaction and equalizing the non-zero off-diagonals result in a smaller graph of the same type. The analysis will be same as for two-dimensional grid graphs. The same analysis holds if the graphs is embedded in three dimensions (or arbitrary dimension d). In this case, the graph edges are edges of tetrahedra (or simplices).

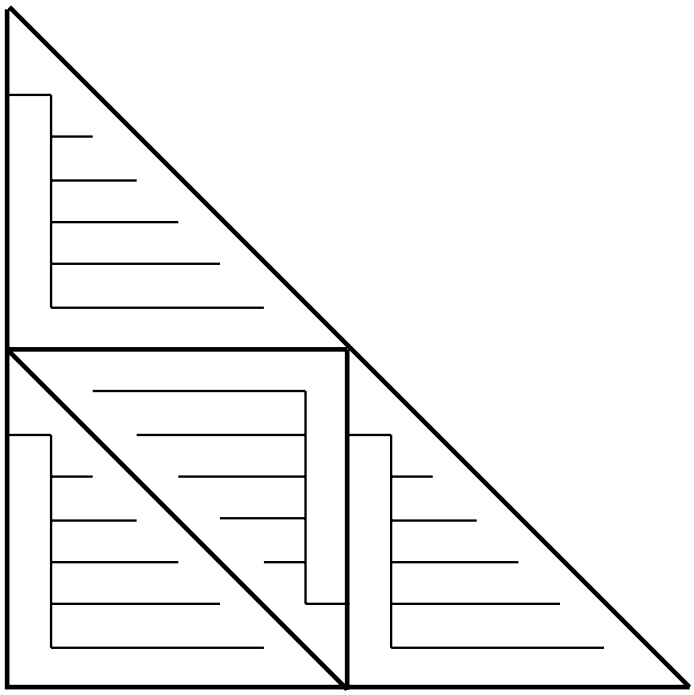


Figure 5.7: Sparsified Recursive Triangular Graph

Recursive Graphs and Other Examples

The same analysis would apply to any graph that has a recursive structure, i.e. after sparsification, compaction and equalizing the edge weights, we get a smaller graph belonging to the same class as the original graph.

The same analysis will apply to problems that are defined on a domain that has been divided into a fixed number of triangles(tetrahedra) and rectangles(parallel piped) of fixed aspect ratio.

5.9 Missing Edges and Boundary Conditions

The procedure of sparsification is the same even when the grid graph has some missing edges as long as the graph has the property that for every node i that has j missing edges (edges which would have been present if it were to be a full grid graph) the diagonal excess of i is such that

$$\Delta_i \geq j\gamma \tag{5.50}$$

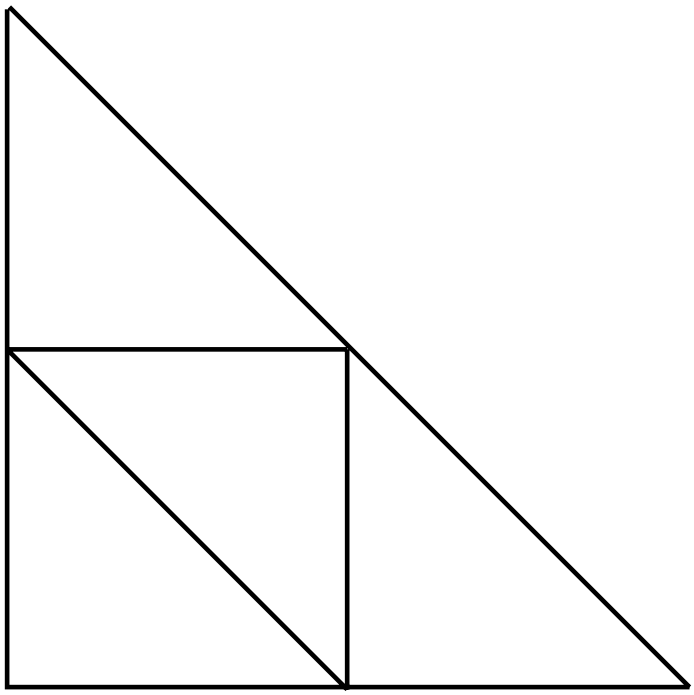


Figure 5.8: Sparsified and Compacted Recursive Triangular Graph

where γ is the value of each conductance.

It is fairly easy to prove that this property is maintained recursively for all the smaller grid graphs that arise in the recursion.

Thus the results of the analysis apply to subgraphs of the grid graphs having the above property. We can use this fact to handle grid graphs that arise from discretizations of continuum problems that have certain boundary conditions imposed on them as follows.

A boundary condition at a node is equivalent to making that node a ground node and removing all the incident edges. This will increase the diagonal excess at each of the neighboring nodes by a value equal to the conductance γ. Thus application of boundary conditions at arbitrary nodes of the grid graph will result in a subgraph of the grid graph which satisfies the property that the diagonal excess at a node i with j missing edges follows Equation 5.50.

5.10 Scope for Further Work

One possible direction of extending the results is to the case where the variation in the non-zero off-diagonals of A may be a function of n. This will be useful in solving arbitrary resistive networks.

Analysis for a different iterative method like Preconditioned Conjugate Gradient (PCG) Method is of interest because the PCG algorithm is simpler than Chebyshev iterative method for the purposes of implementation.

Chapter 6

Conclusions

In the first part of the thesis(Chapters 2,3, and 4), we have investigated the solution of a few optimization problems such as Assignment, Maximum Flow, Minimum Cost Flow, Convex Optimization problems arising out of VLSI timing analysis etc. using interior point algorithms. We have presented numerical results on several real-life problem instances.

One important sub step in the solution of these problems is to solve a large sparse linear system with special properties. In the second part of the thesis(Chapter 5), we have analyzed the quality of a new graph-based preconditioner for Symmetric Positive Definite Diagonally Dominant linear systems that arise out of Network Flow problems, Large Resistive Networks, and Finite Difference discretizations of certain scalar partial differential equations.

BIBLIOGRAPHY

[1] A. Armacost and S. Mehrotra. Computational comparison of the network simplex method with the affine scaling method. *OpSearch*, 28:26–43, 1991.

[2] J. Aronson et al. The projective transformation algorithm of Karmarkar: A computational experiment with assignment problems. Technical Report 85-OR-3, Southern Methodist University, Department of Operations Research,Southern Methodist University, Dallas, Texas, 1985.

[3] D.P. Bertsekas. *Linear Network Optimization: Algorithms and Codes.* The MIT Press, 1991.

[4] D.P. Bertsekas and P. Tseng. RELAX: A computer code for minimum cost network flow problems. *Annals of Operations Research*, 13:127–190, 1988.

[5] D.P. Bertsekas and P. Tseng. Relaxation methods for minimum cost ordinary and generalized network flow problems. *Operations Research*, 36:93–114, 1988.

[6] D.P. Bertsekas and P. Tseng. RELAXT–III: A new and improved version of the RELAX code. Technical Report P-1990, Massachusetts Institute of Technology, Laboratory for Information and Decision Systems, MIT, Cambridge, Massachusetts, 1990.

[7] R. Brayton, E. Detjens, et al. Multiple-level logic optimization system. In *Proceedings, IEEE Intl. Conf. on Computer-Aided Design (ICCAD '86)*, pages 356–359, November 1986.

[8] Björn Breidegard. Design of efficient two-phase clocked systems. Technical report, Dept. of Comp. Eng., Lund Univ., 1991.

[9] E. Brieskorn and H. Knörr. *Plane Algebraic Curves.* Birkhäuser Verlag, Basel, 1986.

[10] A. Charnes. Optimality and degeneracy in linear programming. *Econometrica*, 20:160–170, 1952.

[11] G. Comini et al. *Finite Element Analysis in Heat Transfer:Basic Formulation and Linear Problems*. Computational and Physical Processes in Mechanical and Physical Sciences. Taylor & Francis, Wasshington, D.C., 1994.

[12] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.

[13] M. E. Dagenais and N. C. Rumin. On the calculation of optimal clocking parameters in synchronous circuits with level-sensitive latches. *IEEE Transactions on Computer-Aided Design*, 8(3):268–278, March 1989.

[14] M.E. Dyer. Linear algorithms for two- and three-variable linear programs. *SIAM Journal of Computation*, 13:31–45, 1984.

[15] H. Edelsbrunner. *Algorithms in Combinatorial Geometry*. Springer-Verlag, Berlin, 1987.

[16] Lestor R. Ford Jr. and D.R. Fulkerson. *Flows in Networks*. Princeton University Press, 1962.

[17] R.F. Gabasov and F.M. Kirillova. *Methods of Optimization*. Optimization Software Inc., Publications Division, New York, 1988.

[18] R. Gaur. An implementation of Vaidya's convex programming algorithm. Private Communication, 1993.

[19] A. George. Nested dissection of a regular finite element mesh. *SIAM Journal of Numerical Analysis*, 10:345–363, 1973.

[20] G. H. Golub and C. F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, Baltimore, MD, 1989.

[21] G.H. Golub and M.L. Overton. The convergence of inexact Chebyshev and Richardson iterative methods for solving linear systems. *Numerische Mathematik*, 53:571–593, 1988.

[22] G.H. Golub and R.S. Varga. Chebychev semi-iterative methods, successive over-relaxation iterative methods, and second-order Richardson iterative methods, Parts I and II. *Numerische Mathematik*, 3:147–168, 1961.

[23] L.A. Hageman and D.M. Young. *Applied Iterative Methods.* Academic Press, 1981.

[24] F. S. Hillier and G. J. Liebermann. *Operations Research.* Holden-Day, San Francisco, California, 1974.

[25] A.J. Hoffman, M.S. Martin, and D.J. Rose. Complexity bounds for regular finite difference and finite element grids. *SIAM Journal of Numerical Analysis*, 10:364–369, 1973.

[26] I.Lin et al. Analyzing cycle stealing on synchronous circuits with level-sensitive latches. In *Proceedings, 29th DAC*, pages 393–98, 1992.

[27] G. Jennings. On cycle borrowing analyses for interconnected chips driven by clocks having different but commensurable speeds. In *Proceedings, International Conference on Application-Specific Array Processors (ASAP 92)*, August 4-7 1992. Berkeley, California (to appear).

[28] G. Jennings. On the detection and elimination of superfluous level-sensitive latches. In *Proceedings, Second Great Lakes Symposium on VLSI*, pages 176–182, February 28–29 1992.

[29] G. Jennings and E. Jennings. A formal methodology for level-sensitive latched circuit design with n clocks and m phases. In *Glenn Jennings, Register Transfer Modeling for Design of Latch-Based Synchronous Systems.* Doctoral thesis, Dept. of Computer Engineering, Lund Institute of Technology, Lund, Sweden, May 1992. Paper 7 (in press).

[30] A. Joshi, A. S. Goldstein, and P. M. Vaidya. A fast implementation of a path following algorithm for maximizing a linear function over a network polytope. In *First DIMACS International Workshop on Algorithm Implementation*, Rutgers Univ.,NJ, October 1991.

[31] J.A. Kalinski and Y. Ye. A decomposition variant of the potential reduction algorithm for linear programming. Technical Report 91–11, University of Iowa, Department of Management Sciences, University of Iowa, Iowa City, Iowa, 1991.

[32] N. Karmarkar. A new polynomial algorithm for linear programming. *Combinatorica*, 4(4):373–395, 1984.

[33] J.L. Kennigton and R.V. Helgason. *Algorithms for Network Programming.* John-Wiley & Sons, 1980.

[34] T.A. Manteuffel. The Tchebyshev iteration for nonsymmetric linear systems. *Numerische Mathematik*, 28:307–327, 1977.

[35] N. Megiddo. Linear-time algorithms for linear programming in $\Re^3$ and related problems. *SIAM Journal of Computation*, 12:759–776, 1983.

[36] M. Minoux. *Mathematical Programming: Theory and Practice.* John Wiley and Sons, NY, 1986.

[37] A. Rajan. An empirical comparison of KORBX against RELAXT, a special code for network flow problems. Technical Report (Not available), AT&T Bell Laboratories, Holmdel, NJ, 1989.

[38] J. Renegar. A polynomial-time algorithm, based on Newton's method, for linear programming. *Mathematical Programming*, 40:59–93, 1988.

[39] M.G.C. Resende and G. Veiga. An implementation of the dual affine scaling algorithm for minimum-cost flow on bipartite uncapacitated networks. *SIAM Journal on Optimization*, 3(3):516–537, 1993.

[40] D. J. Rose and G. F. Whitten. A recursive anlaysis of dissection strategies. In J. R. Bunch and D. J. Rose, editors, *Sparse Matrix Computations*, pages 59–84. Academic Press, Inc., New York, 1976.

[41] K. A. Sakallah, T. N. Mudge, and O. A. Olukotun. Analysis and design of latch-controlled synchronous digital circuits. In *Proceedings of the 27th Design Automation Conference*, pages 111–117, June 1990.

[42] K. A. Sakallah, T. N. Mudge, and O. A. Olukotun. checktc and mintc: Timing verification and optimal clocking of synchronous digital circuits. In *Proceedings of the 1990 IEEE Intl. Conference on Computer-Aided Design (ICCAD '90)*, pages 552–555, November 1990.

[43] A. Schrijver. *Theory of Linear and Integer Programming.* John Wiley & Sons, 1986.

[44] P. M. Vaidya. A new algorithm for minimizing convex functions over convex sets. In *Proc. 30th Annual IEEE Symp. Found. Comp. Sc.*, pages 332–337, Oct. 1989.

[45] P. M. Vaidya. A simple Newton-type method for finding the analytical center of a polytope. In Nimrod Megiddo, editor, *Progress in Mathematical Programming : Interior Point and Related Methods*, pages 79–90. Springer-Verlag, New York, 1989.

[46] P. M. Vaidya. An algorithm for linear programming which requires $O(((m+n)n^2 + (m+n)^{1.5}n)L)$ arithmetic operations. *Mathematical Programming*, 47:175–201, 1990.

[47] Pravin M. Vaidya. Solving linear equations with symmetric diagonally dominant matrices by constructing provably good preconditioners. Technical Report 91–1676, Department Of Computer Science, University of Illinois at Urbana-Champaign, 1991.

Vita

Anil Joshi was born in Manthani, Andhra Pradesh, India. He got his B.E. in Electronics and Communications Engineering from Osmania University, Hyderabad in 1980. He got his Post Graduate Diploma in Industrial Engineering from National Institute for Training in Industrial Engineering, Bombay in 1982. He received an M.S. from Polytechnic University of New York, Brooklyn, New York in 1989. He has taken up the US citizenship in 1995.

He has consulted extensively in the areas of Data Base Design and Analysis, Development of Data Dictionaries, Computerization of Accounting, Shipping, Logistics, and Banking in India, Europe and USA during the period of 1982-1989. After completing his Ph.D., he will devote all of his time to growing Computational Applications and System Integration, Inc. (CA&SI), an Illinois corporation, acting as its Vice President and Secretary.