2005

Graph coloring in sparse derivative matrix computation

Goyal, Mini

Lethbridge, Alta. : University of Lethbridge, Faculty of Arts and Science, 2005

http://hdl.handle.net/10133/260

GRAPH COLORING IN SPARSE DERIVATIVE MATRIX COMPUTATION

MINI GOYAL
M.Sc, Banasthali Vidyapith, 2003

A Thesis
Submitted to the School of Graduate Studies
of the University of Lethbridge
in Partial Fulfilment of the
Requirements for the Degree
MASTER OF SCIENCE

Department of Mathematics and Computer Science
University of Lethbridge
LETHBRIDGE, ALBERTA, CANADA

Abstract

There has been extensive research activities in the last couple of years to efficiently determine large sparse Jacobian matrices. It is now well known that the estimation of Jacobian matrices can be posed as a graph coloring problem. Unidirectional coloring by Coleman and Moré [9] and bidirectional coloring independently proposed by Hossain and Steihaug [23] and Coleman and Verma [12] are techniques that employ graph theoretic ideas.

In this thesis we present heuristic and exact bidirectional coloring techniques. For bidirectional heuristic techniques we have implemented variants of largest first ordering, smallest last ordering, and incidence degree ordering schemes followed by the sequential algorithm to determine the Jacobian matrices.

A "good" lower bound given by the maximum number of nonzero entries in any row of the Jacobian matrix is readily obtained in an unidirectional determination. However, in a bidirectional determination no such "good" lower bound is known. A significant goal of this thesis is to ascertain the effectiveness of the existing heuristic techniques in terms of the number of matrix-vector products required to determine the Jacobian matrix. For exact bidirectional techniques we have proposed an integer linear program to solve the bidirectional coloring problem. Part of exact bidirectional coloring results were presented at the "Second International Workshop on Combinatorial Scientific Computing (CSC05), Toulouse, France."

Acknowledgments

I express my deep acknowledgement and profound sense of gratitude to my supervisor Dr. Shahadat Hossain, Assistant Professor, University of Lethbridge, for his inspiring guidance, helpful suggestions and persistent encouragement as well as close and constant supervision throughout the period of my Masters Degree.

I would also like to thank my M.Sc. supervisory committee members Dr. Daya Gaur and Dr. Jim Liu for their guidance and suggestions.

It gives me immense pleasure to acknowledge the financial support from NSERC and the University of Lethbridge Assistantship and Travel Support. I thank all the staff, and my colleagues at the University of Lethbridge for their helpful nature and co-operation.

I dedicate this thesis to my parents, family and Mr. Rahul Jha.

Contents

List of Figures

List of Tables

Symbols

Chapter 1

Introduction

Problems in science and engineering often require to minimize a nonlinear function or to find the numerical solution of a system of nonlinear equations $F(x) = 0$ where $F = (f_1, f_2, ..., f_m)^T$ is a mapping $F : \Re^n \rightarrow \Re^m$. Newton's method (or a variant of Newton's method) can be employed to solve the aforementioned problems [14].

Newton's method is an iterative method which may require a large number of iterations to converge to the solution with desired accuracy. At each iteration one needs to calculate the matrix of first partial derivatives also known as the Jacobian matrix $J(x) \equiv \{\frac{\partial f_i}{\partial x_j}\}, 1 \leq j \leq n, 1 \leq i \leq m$ at the current point x. For very large problems these matrices are often sparse i.e. they contain nonzero entries at very few positions in the matrix, and for complicated functions, computing the Jacobian matrix may dominate the overall computational cost per iteration. Assuming that the sparsity pattern of the matrix is known and it does not change from iteration to iteration, it is important to design efficient methods that take advantage of known sparsity and other structure information like symmetry so that the computations involving known zero entries are avoided in determining the matrix.

The problem of exploiting sparsity in computing the Jacobian matrix can be viewed as a *partitioning problem* [13]. With the known sparsity structure of the given sparse matrix A, we can partition the columns of A into p ($p \leq n$)

groups such that each column belong to exactly one group and the columns in the same group are *structurally orthogonal* i.e. they do not contain more than one nonzero in the same row position. This type of partitioning is called *uni-directional* partitioning and may not be able to exploit the sparsity effectively. Alternately, one can partition the rows and the columns of A simultaneously to obtain p_1 ($p_1 \leq m$) row groups and p_2 ($p_2 \leq n$) column groups. Both of the above partitioning problems can be posed as *graph coloring problems* [10, 12, 17, 23].

Other methods to partition the matrices are column segmenting approach [22, 26, 27, 28] and bidirectional partitioning technique via greedy approach using *distance 3/2 bi-coloring scheme* [18].

The graph coloring problem that we are concerned with in this thesis deals with the assignment of minimum number of positive integers called labels (colors) to the *vertices* of a graph such that no two vertices connected by an *edge* get the same label (color).

Graph coloring plays an important role in a variety of fields of computer science. It models many real-world problems or acts as a part in the overall solution of the problems. Some of the areas where graph coloring is used are register allocation [20], frequency assignment and networks [29], timetabling and scheduling [38], and pattern matching.

In our thesis, the graph coloring problem acts as a tool to determine the Jacobian matrices. By representing the Jacobian matrices as graphs and then partitioning the vertices of the graph using graph coloring, we can partition the rows and columns into groups such that the nonzero entries in each row and column can be solved from a small linear system. This partition information can then be used by *Finite Differencing (FD)* or *Automatic Differentiation (AD)* software to estimate the nonzeros of the Jacobian matrix.

Including this introductory chapter, this thesis contains seven chapters. The outline of the remaining chapters proceeds as follows:

In Chapter 2, we introduce Jacobian matrices followed by the description of Newton's method to solve a system of nonlinear equations and for unconstrained minimization. We then describe unidirectional and bidirectional partitioning techniques, followed by the methods to recover nonzeros. Finally we describe the methods to compute partial derivatives.

In Chapter 3, we provide basic graph theory definitions and notations. We then give the problem definition where we describe graph coloring as related to the partitioning problem. This is followed by a brief description of computational complexities involved with graph coloring, and finally we give the description of graph coloring methods.

In Chapter 4, we feature the existing heuristic techniques for unidirectional and bidirectional p-coloring. We then describe Largest First Ordering (LFO), Smallest Last Ordering (SLO), Incidence Degree Ordering (IDO), and the sequential algorithms as modified by us for bidirectional p-coloring.

In Chapter 5, we introduce exact methods for finding optimal solution of the bidirectional p-coloring. We then explicate a new integer linear programming model for bidirectional p-coloring. Finally we give the computational complexity of the ILP model followed by implementation details.

In Chapter 6, we present experimental results that demonstrate the performance of the algorithms presented in Chapters 4 and 5. We give a comparison of various graph coloring techniques for matrix partitioning. The data for the experiments was provided by the matrix market collection [3].

Finally, in Chapter 7, we provide concluding remarks, as well as possible and proposed directions for future research in this area.

Detailed experimental results are presented in Appendix A and a sample ILP model for bidirectional p-coloring is given in Appendix B. t

Chapter 2

Preliminaries

In this chapter we will identify the problem of determination of sparse Jacobian matrices. In section 2.1 we will introduce Jacobian matrices and give the motivation to determine them. In section 2.2, we will give techniques to partition the Jacobian matrices, followed by section 2.3, in which we will demonstrate methods to recover the nonzeros. In section 2.4, we will describe the methods to compute partial derivatives and finally in section 2.5, we will conclude this chapter.

2.1 Jacobian Matrices

The *Jacobian matrix* is the first-order partial derivative matrix of a vector-valued function. Let $F = (f_1, f_2, ..., f_m)^T$ be a mapping $F : \Re^n \rightarrow \Re^m$. If F is continuously differentiable then the Jacobian matrix of F at x is given by

$$J(x) \equiv F'(x) = \begin{pmatrix} \frac{\partial}{\partial x_1} f_1(x) & \cdots & \frac{\partial}{\partial x_n} f_1(x) \\ \vdots & & \vdots \\ \frac{\partial}{\partial x_1} f_m(x) & \cdots & \frac{\partial}{\partial x_n} f_m(x) \end{pmatrix}. \tag{2.1}$$

Derivative information is needed, for example in the solution of systems of nonlinear equations and in the unconstrained minimization problems. Newton's methods are one of the classical methods to solve the systems of nonlinear equations and to obtain unconstrained minimization respectively.

2.1.1 Newton's Method for Systems of Nonlinear Equations

Given $F : \Re^n \rightarrow \Re^m$, the solution to the associated system of nonlinear equations is attained by finding $x^{\{*\}} \in \Re^n$ such that $F(x^{\{*\}}) = 0$ where F is assumed to be continuously differentiable [14]. In inexact Newton's method, the solution of the resulting linear system is approximated by a linear iterative method. Following are the steps for solving this nonlinear system.

Algorithm 1 Newton's Method for Systems of Nonlinear Equations

Let $x^{\{0\}} \in \Re^n$;

for $j \leftarrow 0$, convergence **do**

$J(x^{\{j\}})s^{\{j\}} = -F(x^{\{j\}})$;

▷ Compute the Jacobian matrix at current point and solve for step $s^{\{j\}}$

$x^{\{j+1\}} = x^{\{j\}} + s^{\{j\}}$; ▷ Update the current point

end for

$J(x)$ is known as the Jacobian matrix of F at x.

The following example illustrates Newton's Method to solve the systems of nonlinear equations.

Given

$$F(x) = \begin{bmatrix} x_1 + x_2 - 3 \\ x_1^2 + x_2^2 - 5 \end{bmatrix}$$

with roots at $(1, 2)^T$ and $(2, 1)^T$.

The Jacobian matrix is given by

$$\begin{bmatrix} 1 & 1 \\ 2x_1 & 2x_2 \end{bmatrix}.$$

Let $x^{\{0\}} = (0, 3)^T$. Then the first two iterations of Newton's method are

$$J(x^{\{0\}})s^{\{0\}} = -F(x^{\{0\}}) : \begin{bmatrix} 1 & 1 \\ 0 & 6 \end{bmatrix} s^{\{0\}} = -\begin{bmatrix} 0 \\ 4 \end{bmatrix}, \text{ gives } s^{\{0\}} = \begin{bmatrix} \frac{2}{3} \\ -\frac{2}{3} \end{bmatrix},$$

$$x^{\{1\}} = x^{\{0\}} + s^{\{0\}} = (0.667, 2.333)^T,$$

$$J(x^{\{1\}})s^{\{1\}} = -F(x^{\{1\}}) : \begin{bmatrix} 1 & 1 \\ \frac{4}{3} & \frac{14}{3} \end{bmatrix} s^{\{1\}} = -\begin{bmatrix} 0 \\ \frac{8}{9} \end{bmatrix}, \text{ gives } s^{\{1\}} = \begin{bmatrix} \frac{4}{15} \\ -\frac{4}{15} \end{bmatrix},$$

$$x^{\{2\}} = x^{\{1\}} + s^{\{1\}} \cong (0.933, 2.067)^T.$$

If the initial approximation $x^{\{0\}}$ is sufficiently close to the root, it is expected that the successive iterates will converge to the root.

2.1.2 Newton's Method for Unconstrained Minimization

Another important problem from optimization where the derivative information is required is the unconstrained minimization problem

$$\min_{x \in \Re^n} f : \Re^n \to \Re, \tag{2.2}$$

where f is assumed to be twice continuously differentiable. The algorithm for Newton's method for unconstrained minimization is given as follows.

Algorithm 2 Newton's Method for Unconstrained Minimization

Let $x_0 \in \Re^n$;

for $j \leftarrow 0$, minimization **do**

$\nabla^2 f(x_j) s_j^N = -\nabla f(x_j)$,

$x_{j+1} = x_j + s_j^N$. ▷ Update the current point

end for

Here $\nabla^2 f(x)$ is the Hessian matrix and $\nabla f(x)$ is the gradient of f. The Hessian of f can be viewed as the Jacobian of $\nabla f(x)$.

At every iteration of Newton's method we need to determine the Jacobian matrix at the current point. In many large problems the Jacobian matrix is sparse i.e. there are very few nonzeros in the matrix. By exploiting this sparsity, we can efficiently determine the Jacobian matrix and thus significantly

reduce the overall computational cost of the solution process. In the next section we will discuss methods to partition the Jacobian matrices.

2.2 Matrix Partitioning

In 1974, Curtis, Powell and Reid [13] noted that the sparsity of the Jacobian matrices can be exploited if the columns of the matrix can be partitioned into groups such that columns in each group are structurally orthogonal to each other.

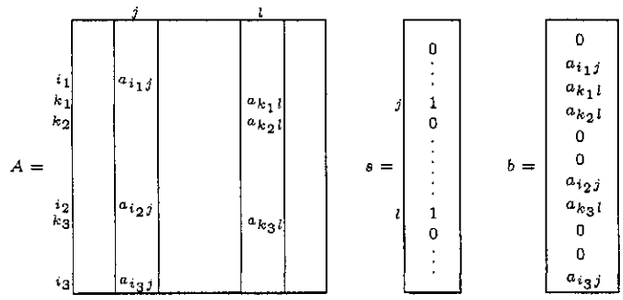


Figure 2.1: Example by Curtis, Powell and Reid

Let $A \in \Re^{m \times n}$ be the given matrix. In Figure 2.1 we see that columns j and l of A are *structurally orthogonal* i.e. there does not exist a row index i for which both $a_{ij} \neq 0$ and $a_{il} \neq 0$. The corresponding vector s is initialized as $\sum_j e_j$, where e_j is the jth coordinate vector and the sum for this vector is taken over a set of structurally orthogonal columns. Vector b is obtained as the product $b = As$ by using FD or AD forward mode. We see that b contains the unique nonzero entries of columns j or l (or a zero) at each position. More generally, consider structurally orthogonal partitioning of A into p groups. We can then define a *seed matrix* $S \in \Re^{n \times p}$ where each column of S corresponds to a group of structurally orthogonal columns and is defined by $\sum_j e_j$ as discussed

earlier. Then the nonzeros of A can be recovered from the product $B = AS$ obtained through forward automatic differentiation or finite differencing.

2.2.1 Unidirectional Partitioning

A partitioning scheme in which either the columns or the rows are partitioned into structurally orthogonal groups is known as *unidirectional partitioning.*

As shown in Figure 2.2, matrix A can be partitioned into two column groups such that all the nonzeros of A can be obtained from the product AS.

$$A = \begin{bmatrix} \times & & & & \\ \times & \times & & & \\ \times & & \times & & \\ \times & & & \times & \\ \times & & & & \times \end{bmatrix}, \quad S = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 1 \\ 0 & 1 \\ 0 & 1 \end{bmatrix}$$

Figure 2.2: Column Partitioning

In Figure 2.3, we see that by partitioning the matrix A into two row groups, we can obtain all the nonzeros of A from the product $W^T A$.

$$A = \begin{bmatrix} \times & \times & \times & \times & \times \\ & \times & & & \\ & & \times & & \\ & & & \times & \\ & & & & \times \end{bmatrix}, \quad W^T = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 \end{bmatrix}$$

Figure 2.3: Row Partitioning

2.2.2 Bidirectional Partitioning

For a given matrix $A \in \Re^{m \times n}$, if seed matrices $S \in \Re^{n \times p_1}$ and $W \in \Re^{m \times p_2}$ can be obtained such that all the nonzeros of A can be determined uniquely from the products $B = AS$ and $C^T = W^T A$, then the resulting partitioning is known as *bidirectional partitioning.*

Considering Figure 2.4, we notice that unidirectional partitioning (either row or column) will require at least 5 groups. But if we determine row 1 and column 1 separately and collect the remaining nonzeros in one column(row) group then we require only 3 groups.

$$A = \begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & & & \\ \times & & \times & & \\ \times & & & \times & \\ \times & & & & \times \end{bmatrix}, S = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 1 \\ 0 & 1 \\ 0 & 1 \end{bmatrix}, W^T = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Figure 2.4: Bidirectional Partitioning

2.3 Methods for Recovering Nonzeros

In this section we briefly describe the techniques to recover the nonzeros from the product $B = AS$, where A is the Jacobian matrix to be determined.

For a given matrix $A \in \Re^{m \times n}$, we want to obtain seed matrices $S \in \Re^{n \times p_1}$ and $W^T \in \Re^{p_2 \times m}$ such that all the nonzeros of A can be determined from the products $B = AS$ and $C^T = W^T A$.

In the following we outline a procedure for unidirectional determination of a Jacobian matrix $A \in \Re^{m \times n}$.

- Obtain $B = AS$ as p matrix-vector products using finite differencing or forward automatic differentiation.
- Identify the reduced seed matrix as $\hat{S}_i \in \Re^{\rho_i \times p}$, where ρ_i is the number of nonzeros in row i of A.
- Solve for the nonzeros in row i of A

$$\hat{S}_i^T \alpha = \beta \tag{2.3}$$

where α contains the nonzero unknowns in row i and β is the corresponding vector in matrix B.

If, for every row of A the reduced system (2.2) is a permuted identity matrix then we have a *direct method* [23]. If the reduced system can be permuted to a triangular system then we have a *substitution method* [24], otherwise we have an *elimination method* [25].

2.3.1 Direct Method

In direct determination method, all the nonzeros of A can be read-off from the matrix B and C^T without any further arithmetic operation. Let us demonstrate the direct determination method with the help of the following example. Let

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\ a_{21} & a_{22} & 0 & 0 & 0 \\ a_{31} & 0 & a_{33} & 0 & 0 \\ a_{41} & 0 & 0 & a_{44} & 0 \\ a_{51} & 0 & 0 & 0 & a_{55} \end{bmatrix}, S = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 1 \\ 0 & 1 \\ 0 & 1 \end{bmatrix}, W^T = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

Thus we can obtain the matrices B and C^T by the matrix-vector product AS and vector-matrix product $W^T A$ respectively.

$$B = \begin{bmatrix} a_{11} & a_{12} + a_{13} + a_{14} + a_{15} \\ a_{21} & a_{22} \\ a_{31} & a_{33} \\ a_{41} & a_{44} \\ a_{51} & a_{55} \end{bmatrix}, C^T = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \end{bmatrix}.$$

The nonzeros of A can thus be read off from B and C^T.

2.3.2 Substitution Method

In a substitution method the unknown elements of the matrix A are determined by solving a triangular system of equations i.e. the ordering of the nonzeros of A is such that every nonzero is determined using formerly computed values. Let us comprehend this method with the help of an example illustrated in [24]. Let

$$A = \begin{bmatrix} a_{11} & 0 & a_{13} \\ a_{21} & a_{22} & 0 \\ 0 & a_{32} & a_{33} \end{bmatrix}, \text{ and let } S = \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 1 \end{bmatrix}.$$

The second row of A can be determined by solving for a_{21} and a_{22} in the following reduced system

$$\begin{bmatrix} a_{21} & a_{22} & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} b_{21} & b_{22} \end{bmatrix}.$$

Eliminating row 3 of S and transposing the system, we get

$$\begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} a_{21} \\ a_{22} \end{bmatrix} = \begin{bmatrix} b_{21} \\ b_{22} \end{bmatrix},$$

which is an upper triangular system. The nonzeros of the other two rows of A can be found in the similar way. Substitution method usually require fewer number of function evaluation or AD passes but is subject to numerical instability.

It can be verified that the above example cannot be determined with fewer than 3 matrix-vector products in any direct methods.

2.3.3 Elimination Method

Elimination method is a general method where no special structure is assumed for a seed matrix. Any square submatrix of the seed matrix S, however, must

be nonsingular. Let us view this method with the help of the following example. Let,

$$A = \begin{bmatrix} a_{11} & 0 & a_{13} & a_{14} & 0 \\ 0 & a_{22} & a_{23} & 0 & a_{25} \\ a_{31} & a_{32} & 0 & a_{34} & 0 \end{bmatrix}.$$

The successive column merging technique [25] gives the following seed matrix

$$S = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 1 & 2 & 1 \\ 0 & 1 & 2 \\ 0 & 0 & 1 \end{bmatrix}.$$

The matrix B could be obtained by the product $B = AS$, giving

$$B = \begin{bmatrix} a_{11} + a_{13} & 2a_{13} + a_{14} & a_{13} + 2a_{14} \\ 2a_{22} + a_{23} & a_{22} + 2a_{23} & a_{23} + a_{25} \\ a_{31} + 2a_{32} & a_{32} + a_{34} & 2a_{34} \end{bmatrix}.$$

Then the unknowns for example in row 1 of A can be determined as follows

$$\begin{bmatrix} 1 & 1 & 0 \\ 0 & 2 & 1 \\ 0 & 1 & 2 \end{bmatrix} \begin{bmatrix} a_{11} \\ a_{13} \\ a_{14} \end{bmatrix} = \begin{bmatrix} b_{11} \\ b_{12} \\ b_{13} \end{bmatrix}.$$

2.4 Computing Partial Derivatives

2.4.1 Finite Difference Approximation

Let A denote the Jacobian matrix $J(x)$ of a continuously differentiable mapping $F : \Re^n \rightarrow \Re^m$. An approximation to the jth column of A, denoted by a_j, can be obtained from

$$a_j = \frac{\partial}{\partial x_j} F(x) \approx \frac{1}{\varepsilon}[F(x + \varepsilon e_j) - F(x)], 1 \leq j \leq n, \tag{2.4}$$

where e_j is the jth coordinate vector and ε is a positive increment. Assuming $F(x)$ has already been evaluated, we can estimate the partial derivatives in the jth column of matrix A through the additional function evaluation $F(x+\varepsilon e_j)$. Note that, if the sparsity information is not exploited then we will need n extra function evaluations to determine A.

The advantage of finite difference is that it is easy to implement. The finite difference method can be used as a black box i.e, to obtain an approximation to the derivatives, we do not need to access the function code. We just need to call the subroutine that implements the mathematical function. The disadvantage of finite differencing is that it is prone to numerical instability. If ε is taken to be too large then the approximation is not accurate due to truncation error and if ε is taken to be too small then $F(x + \varepsilon e_j) - F(x)$ may cause loss of precision to round-off errors associated with finite precision calculations.

2.4.2 Automatic Differentiation

Automatic Differentiation (AD) is a chain rule based technique for evaluating the derivatives of functions defined by computer programs. Unlike finite difference approximation (FD), the derivatives computed using AD are free from truncation errors. We will now present a brief description of basic AD techniques. For a comprehensive introduction to AD we refer to the excellent reference [19] by Andreas Griewank.

A program for evaluating the function

$$z = F(x), F : \Re^n \rightarrow \Re^m \tag{2.5}$$

can be seen as a series of scalar assignments

$$v_i = \phi_i(v_j)_{j \rightarrow i}, \tag{2.6}$$

where $j \rightarrow i$ indicates that v_j is computed before v_i. Variables v_j are ordered

such that they can be divided into three vectors:

$$x \equiv (v_1, v_2, ..., v_n)^T \text{ (independent variables)},$$

$$y \equiv (v_{n+1}, v_{n+2}, ...v_{n+p})^T \text{ (intermediate variables)},$$

$$z \equiv (v_{n+p+1}, v_{n+p+2}, ..., v_{n+p+m})^T \text{ (dependent variables)}. \tag{2.7}$$

ϕ_i represent *elementary functions*, which can be arithmetic operations or transcendental functions. If all these elementary functions ϕ_i are well defined and have continuous *elementary partials*

$$c_{ij} = \frac{\partial}{\partial v_j}\phi_i, j < i, \tag{2.8}$$

then by the repeated application of the chain rule, the nonzeros of the Jacobian matrix $J(x)$ can be computed from the elementary partials c_{ij}. AD has two basic modes of operation namely *forward* and *reverse*.

Forward Mode

In forward mode, intermediate partial derivatives are accumulated in the same order as the function values are computed. A *forward pass* is equivalent to the calculation of the matrix vector product Jv where v is a n-vector. By initializing v to be unit coordinate vector $e_i, i = 1, 2, ..., n$, all the columns of J can be determined by n forward passes.

Reverse Mode

In reverse mode, the intermediate partial derivatives are accumulated in reverse order of function evaluation. A *reverse pass* corresponds to the computation $w^T J$ where w is a m-vector. By initializing w to be unit coordinate vectors $e_i, i = 1, 2, ..., m$ all the rows of J can be determined by m reverse passes.

In the above descriptions we noticed that the nonzeros of A can be efficiently determined from B and C^T. By obtaining seed matrices $S \in \Re^{n \times p_1}$

and $W^T \in \Re^{p_2 \times m}$ such that p_1 and p_2 is minimized, we can reduce the number of function evaluations in FD and the number of forward and reverse passes in AD, thus minimizing the computational cost of determining the Jacobian matrix.

2.5 Summary

In this chapter we discussed numerical algorithms where efficient computation of partial derivatives is crucial. We introduced unidirectional and bidirectional partitioning that exploits sparsity and used examples illustrating different techniques to "recover" the nonzero entries from the products AS and $W^T A$. We briefly described FD and AD techniques to obtain approximation to the nonzero entries. In the next chapter we will present graph coloring technique to partition the Jacobian matrices.

Chapter 3

Background

In this chapter we will give the problem definition and all the pertinent terminology that will be used in this and the subsequent chapters. In section 3.1 we will give graph notations followed by section 3.2 in which we will define the problem of bipartitioning the Jacobian matrix using graph coloring. We will discuss the complexity issues associated with bidirectional p-coloring in section 3.3 and in section 3.4 we will describe the graph coloring methods. Finally, in section 3.5 we will summarize this chapter.

3.1 Graph Theoretic Definitions and Notations

A *graph* G is an ordered pair (V, E) where V is a finite and nonempty set called *vertices* and E is a set of unordered pairs of distinct vertices called *edges*. Two vertices u and v are *adjacent* if and only if $\{u, v\} \in E$. The *degree* of a vertex v is the number, denoted $\deg(v)$, of edges with v as an endpoint. A *path* $\mathcal{P}$ of *length* l is a sequence $\{v_1, v_2, ..., v_{l+1}\}$ of distinct vertices in G such that v_i is adjacent to v_{i+1}, for $1 \leq i \leq l$.

A *bipartite graph* $G_b = (U \cup V, E)$ contains two disjoint sets of vertices U and V such that every edge in G has adjacent vertices in U and V respectively.

3.2 Problem Definition

3.2.1 Graph Coloring

Graph coloring is an assignment of colors or labels to the vertices of the graph such that no two adjacent vertices receive the same color.

A *p-coloring* of a graph $G = (V, E)$ is a function $\phi : V \rightarrow \{1, 2, ..., p\}$ such that $\phi(u) \neq \phi(v)$ if $\{u, v\} \in E$. The *chromatic number* $\chi(G)$ is the smallest p for which G has a p-coloring. A coloring that uses $\chi(G)$ colors is known as *optimal coloring.*

Figure 3.1 illustrates p-coloring of the graph G using $p = 3$ colors.

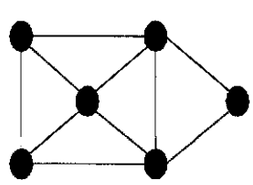


Figure 3.1: p-coloring Example

3.2.2 Formulating the Partitioning Problem as a Graph Coloring Problem

Direct determination as proposed in this thesis is based on partitioning the rows and columns of the Jacobian matrix such that the nonzero entries can be recovered from the matrix-vector products computed via AD or FD. We can conveniently reformulate the partitioning problem as a coloring problem of an associated graph. In this section we will discuss graph formulation of the partitioning problem.

Consider a $m \times n$ matrix A. The column intersection graph of A is a graph $G(A) = (V, E)$ where for each column $j, j = 1, 2, ..., n$ of A there is a vertex

$v \in V$ and $\{v_k, v_l\} \in E$ if there is a row i such that $a_{ik} \neq 0$ and $a_{il} \neq 0$. Figure

(a) A (b) $G(A)$

Figure 3.2: Sparse Matrix and its Column Intersection Graph Representation

3.2(a) depicts the matrix A and Figure 3.2(b) represents its corresponding column intersection graph.

The following result [9, 35] states the connection between the unidirectional partitioning problem and graph coloring.

Theorem 3.1 ϕ is a unidirectional partitioning of the columns (or rows) of A if and only if ϕ induces a coloring of the graph $G(A)$ (or $G(A^T)$).

As has been observed in [12, 23], unidirectional partitioning may not yield the most effective exploitation of matrix sparsity. In the unidirectional partitioning the graph defined for a sparse matrix A represents the sparsity of either columns $G(A)$ or rows $G(A^T)$ but not both. To represent both row and column sparsity a different graph is needed. Specifically, we need to record the zero-nonzero structure of rows and columns. A bipartite graph is a convenient data structure for this purpose.

The bipartite graph associated with matrix $A \in \Re^{m \times n}$ is a graph $G_b(A) = (U \cup V, E)$ where $U = \{u_1, u_2, ...u_m\}, V = \{v_1, v_2, ..., v_n\}$ and $\{u_i, v_j\} \in E$ whenever a_{ij} is a nonzero element of A, for $1 \leq i \leq m, 1 \leq j \leq n$. The size of the graph $G_b(A)$ is proportional to the size of the matrix A, such that the number of vertices $|U| + |V| = m + n$ and number of edges $|E| = nnz(A)$, where $nnz(A)$ is the number of nonzeros in A.

(a) A (b) $G_b(A)$

Figure 3.3: Sparse Matrix and its Bipartite Graph Representation

Figure 3.3(a) shows a sparse matrix A and Figure 3.3(b) represents its associated bipartite graph.

A mapping $\phi : U \cup V \rightarrow \{1, 2, ..., p\}$ is called a *bidirectional p-coloring* of bipartite graph $G_b = (U \cup V, E)$ if ϕ is p-coloring of G_b and every path of length 3 in G_b uses at least 3 different colors such that

$$\{\phi(u) : u \in U\} \cap \{\phi(v) : v \in V\} = \emptyset. \tag{3.1}$$

The chromatic number for bidirectional p-coloring is denoted by $\chi(G_b)$.

It has been proved that bidirectional partitioning of A is equivalent to bidirectional p-coloring of $G_b(A)$ [12, 23].

3.3 Intractability

Computational complexity is concerned with analyzing the resources needed to solve computational problems. Complexity theory is used as a tool to analyze algorithms, so that the *bounds* on the resources required for solving the computational problem can be determined.

A *decision problem* is one whose solution is either "yes" or "no". A decision problem π for which the answer (yes or no) can be decided in polynomial time i.e. the worst case running time for an algorithm solving the problem π is $O(n^k)$, where n is the size of the inputs and k is some constant, then that

problem is said to be in the class P [16, 30]. The problems within class P are known as *tractable.* If k is sufficiently small then these problems can be solved in a reasonable amount of time.

A decision problem π for which a solution can be guessed and verified in polynomial time belongs to the class NP. Some problems in NP are shown to be the members of the equivalence class NP-complete (NPC). A decision problem π belongs to the class NPC if $\pi \in$ NP and for every other problem π' in NP there exists a polynomial time algorithm that transforms π' to π such that if the solution to π is 'yes' then the solution to π' is also 'yes' and viceversa. The problems in class NPC are most difficult to solve and no algorithm to date is known which can solve these problems in deterministic polynomial time.

A *Combinatorial optimization problem* (COP) is either a "minimization problem" or a "maximization problem". For each instance I of a problem, there exists a finite set $S(I)$ of "candidate solutions" for I. A function m is called a "solution value" for each candidate solution if it assigns to each instance and each candidate solution a rational number. In a minimization (maximization) problem, an *optimal solution* for an instance I is a candidate solution $\sigma*$ such that for all possible candidate solutions, $\sigma*$ has the minimum (maximum) solution value.

The optimization version of the decision problems in NPC belong to the class *NP-hard* i.e. a problem is considered as hard as NPC. Also no algorithm to date exist which can solve NP-hard problems in polynomial time. The class of NPC and NP-hard are regarded as *intractable* because problems in these classes have no known polynomial time algorithms.

In our thesis we are concerned with the optimization version of the coloring and partitioning problems (unidirectional and bidirectional).

3.4 Graph Coloring Methods

We can apply both heuristic techniques as well as exact methods to color the vertices of the graph. We have applied heuristic techniques to solve the partitioning problem because they are solvable in polynomial time and give good solutions but we want to know how good are the heuristics doing and this has motivated us to investigate exact coloring techniques. We will give a short description of both the techniques below.

3.4.1 Heuristic Methods

Algorithms which give solution in given time, and do not guarantee any upper or lower bounds but they often find "good" solutions are called *heuristics* or *inexact* methods. The performance measurement for these methods is usually done by benchmarking i.e. measuring the quality of performance on different sets of inputs. The weakness of this performance measuring is that it is difficult to predict the results of arbitrary sets of inputs. In our thesis we have adapted three well-known heuristic algorithms namely largest first ordering (LFO), smallest last ordering (SLO) and incidence degree ordering (IDO) for bidirectional p-coloring.

3.4.2 Exact Methods

Algorithms that give optimal solution for the given problem are known as *exact* methods. These algorithms give upper and lower bounds of the problems and confirm that no better solution could be found. Exact methods are "hard" and often not solvable in polynomial time. In our thesis we have formulated an integer linear programming (ILP) model to implement the bidirectional p-coloring.

3.5 Summary

In this chapter we introduced the notations as used in our thesis. We defined graph coloring and discussed the formulation of the partitioning problem as a graph coloring problem. We reviewed unidirectional and bidirectional p-coloring schemes. We presented intractability and described heuristic and exact graph coloring methods. In the next chapter we will discuss heuristic algorithms for bidirectional p-coloring.

Chapter 4

Coloring Heuristics

In this chapter we will study heuristic techniques to determine the sparse Jacobian matrices. In section 4.1 we will discuss existing unidirectional and bidirectional heuristic techniques, in section 4.2 we will detail heuristic techniques developed for bidirectional graph coloring, and finally in section 4.3 we will summarize the chapter.

4.1 Background

4.1.1 Unidirectional Graph Coloring

In 1983, Coleman and Moré [10] suggested that the column partitioning problem could be posed as a graph coloring problem. They proposed algorithms in which they ordered the vertices of the column intersection graph $G(A)$ using the largest first ordering (LFO), smallest last ordering (SLO), and incidence degree ordering (IDO) schemes, and then applied the sequential algorithm on these ordered vertices [9].

4.1.2 Bidirectional Graph Coloring

Unidirectional coloring deals with either the rows or columns of the sparse matrix A while bidirectional coloring involves both rows and columns of A. As

discussed in section 2.3 it is desirable to minimize p such that all the nonzeros of A are determined uniquely. The following subsections discuss existing heuristic techniques for bidirectional p-coloring.

Complete Direct Cover

Hossain and Steihaug [23] proposed *row-column consistent partitioning* of A in which the entire set of rows and columns is partitioned. They introduced *complete direct cover* for Jacobian matrices as described below.

Let S_c be a collection of subsets of columns and S_r be a collection of subset of rows. The set $\{S_c, S_r\}$ is called complete direct cover of A if

- The intersection of any two subsets is empty.
- For each nonzero element a_{ij}, there is a subset $X \in S_c \cup S_r$ such that a_{ij} is directly determined by X.

An algorithm to compute complete direct cover aims to find groups of rows and columns that satisfy the direct cover property. The algorithm terminates when all the nonzeros are determined. Maximum number of colors needed to determine Jacobian matrix directly using complete direct cover algorithm is $|S_c| + |S_r| + 2$ [23].

Bicoloring

Coleman and Verma [11, 12] studied the same problem and suggested that it is sufficient to partition subsets of rows and columns such that A is determined directly. The vertices that are not involved in the determination of any nonzero entry are assigned the *neutral color zero*. The bipartite coloring scheme applied by them is illustrated below.

Let $G_b = (U \cup V, E)$ be a bipartite graph. The mapping $\phi : U \cup V \rightarrow \{0, 1, ..., p\}$ is a bipartite p-coloring of G_b if the following conditions hold.

- If $u \in U$ and $v \in V$, then $\phi(u) \neq \phi(v)$ or $\phi(u) = \phi(v) = 0$.
- If $\{u, v\} \in E$, then $\phi(u) \neq 0$ or $\phi(v) \neq 0$.
- If vertices u and v are adjacent to vertex w with $\phi(w) = 0$, then $\phi(u) \neq \phi(v)$.
- Every path of three edges uses at least three colors.

They introduced the concept of *bicoloring* in which A is permuted and partitioned. Minimum nonzero count ordering (MNCO) algorithm is built to partition J to obtain matrix J_c from bottom up and J_r from right to left. At every iteration in MNCO either a new column is added to J_c or a new row is added to J_r. The coloring is then obtained by partitioning the columns in J_c and partitioning the rows in J_r. This double coloring approach is named as bicoloring.

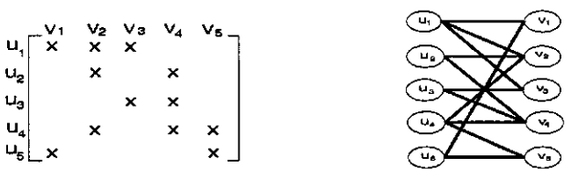
4.2 Bidirectional Heuristic Techniques

In this section we will discuss our bidirectional heuristic techniques. We initially order the vertices of the bipartite graph $G_b(A)$ using one of largest first ordering (LFO), smallest last ordering (SLO), and incidence degree ordering (IDO). We then apply sequential algorithm on the ordered vertices to obtain bidirectional p-coloring of $G_b(A)$.

From Figure 4.1, the degrees of row and column vertices can be enumerated as follows.

$deg(u_1) = 3, deg(u_2) = 2, deg(u_3) = 2, deg(u_4) = 3, deg(u_5) = 2$

$deg(v_1) = 2, deg(v_2) = 3, deg(v_3) = 2, deg(v_4) = 3, deg(v_5) = 2$



(a) A (b) $G_b(A)$

Figure 4.1: Sparse Matrix A and its Bipartite Graph $G_b(A)$

We will illustrate the algorithms using the example matrix given in Figure 4.1.

4.2.1 Largest First Ordering

In largest first ordering (LFO) we first sort the vertices in U and V of the bipartite graph $G_b(A)$ in nonincreasing order of their degrees such that $deg(u_1) \geq ... \geq deg(u_m)$ and $deg(v_1) \geq ... \geq deg(v_n)$. The two sets of sorted vertices are then merged into one ordering.

Algorithm 3 depicts the sorting routine applied in Algorithm 4 to sort the row vertices. The same routine is applied to sort the column vertices also. In Algorithm 3, ρ_{max} and ρ_{min} represent the maximum and minimum number of nonzeros in any row or column of A, respectively. The array ndegr holds the degree of row vertices such that ndegr(i) is the degree of row i of A.

In Algorithms 3 and 4, RowDeg represents the array containing the indices of the row vertices in nonincreasing order of their degrees and ColDeg represents the array containing the indices of the column vertices in nonincreasing order of their degrees. The arrays RowDeg and ColDeg computed by Algorithm 3 for the matrix given in Figure 4.1(a) is as follows.

RowDeg

index	1	2	3	4	5
U	**1**	**4**	**2**	**3**	**5**

ColDeg

index	1	2	3	4	5
V	**2**	**4**	**1**	**3**	**5**

In Algorithm 3, while sorting the vertices of $G_b(A)$, we take advantage of prior knowledge of matrix structure.

Algorithm 3 Sorting Algorithm

```
1:  procedure SORTING(Rows)
2:      icr = 0;
3:      for j ← ρ_max, ρ_min do
4:          for i ← 1, m do
5:              if ndegr(i) == j then
6:                  RowDeg(icr) = i;
7:                  icr + +;
8:              end if
9:          end for
10:     end for
11: end procedure
```

The outer *for* loop at line 3 runs $(\rho_{max} - \rho_{min})$ times and the inner *for* loop at line 4 runs m times, implying that the running time of the sorting algorithm is $O(m(\rho_{max} - \rho_{min}))$. Since $\rho_{min} > 0$, this sorting runs in $O(m\rho_{max})$ time.

In Algorithm 4, arrays ListRow and ListCol contain the ordering information of row and column vertices respectively and together they determine the combined ordering in which the vertices are processed by the sequential algorithm. ListCol(i) denotes that column vertex v_i will be processed by the sequential ordering algorithm after the vertices that are ordered before v_i in largest first ordering. This combined ordering is computed by the statements on lines 6-15. The counter *inc* is incremented by one at each iteration of the while loop.

Algorithm 4 Largest First Ordering

```
1: procedure LFO(G_b(A))
2:     inc ← 1;
3:     icr ← 1, icc ← 1;
4:     Sort the vertices in U in nonincreasing order of their degrees and put
           the result in RowDeg;
5:     Sort the vertices in V in nonincreasing order of their degrees and put
           the result in ColDeg;
6:     while inc ≤ m + n do                ▷ Ordering row and column vertices
7:         if Degree of vertex at RowDeg(icr) ≥ Degree of vertex at
               ColDeg(icc) then
8:             ListRow(RowDeg(icr)) ← inc;
9:             icr + +;
10:        else
11:            ListCol(ColDeg(icc)) ← inc;
12:            icc + +;
13:        end if
14:        inc + +;
15:    end while
16: end procedure
```

The combined ordering computed by LFO for matrix in Figure 4.1(a) is shown below.

ListRow

U	1	2	3	4	5
Ordering	**1**	**5**	**6**	**2**	**7**

ListCol

V	1	2	3	4	5
Ordering	**8**	**3**	**9**	**4**	**10**

In Algorithm 4, the running time for sorting of row vertices is $O(m\rho_{max})$ and that of column vertices is $O(n\kappa_{max})$. Statements on lines 6-15 orders the vertices in $O(m+n)$. Thus the total running time for the largest first ordering

algorithm is $O(\max\{m\rho_{max}, n\kappa_{max}, (m+n)\})$ which is $O(\max\{m\rho_{max}, n\kappa_{max}\})$. Without loss of generality, let $m = \max\{m, n\}$ and $\rho_{max} = \max\{\rho_{max}, \kappa_{max}\}$, then the run time complexity of LFO algorithm is $O(m\rho_{max})$.

Before we examine the smallest last ordering and incidence degree ordering, we will require additional graph terminology. Given a graph $G_b = (U \cup V, E)$ and a nonempty subset U_1 of U and V_1 of V, the *subgraph* $G_{b_1}[U_1 \cup V_1]$ *induced* by $U_1 \cup V_1$ has the vertex set $U_1 \cup V_1$ and the edge set

$$\{\{u, v\} \in E : u \in U_1, \text{ and } v \in V_1\}.$$

4.2.2 Smallest Last Ordering

In smallest last ordering (SLO) the row or column vertex chosen at the kth stage has minimal degree in the graph induced by the unordered vertices i.e. kth vertex w_k is determined after $w_{k+1}, w_{k+2}, ..., w_{m+n-1}, w_{m+n}$, where w_i is either a row vertex u_j or a column vertex v_l, have been selected by choosing w_k so that its degree in the subgraph induced by

$$(U \cup V) \setminus \{w_{k+1}, w_{k+2}, ..., w_{m+n-1}, w_{m+n}\}$$

is minimal.

In Algorithm 5, *inc* is the ordering counter which starts from $m+n$. Arrays ListRow and ListCol, as described for LFO, store the ordering information of row and column vertices respectively. Lines 3 and 4 find the minimum degree row vertex u_{mindeg} and column vertex v_{mindeg}. Lines 7 and 14 decrease the degrees of the vertices adjacent to u_{mindeg} and v_{mindeg} respectively. Lines 8 and 15 order the minimum degree vertex and lines 11 and 17 recompute v_{mindeg} and u_{mindeg} among the remaining unordered vertices.

Algorithm 5 Smallest Last Ordering

```
1: procedure SLO(G_b(A))
2:     inc ← m + n;
3:     Find u_mindeg ← minimum degree row vertex in U;
4:     Find v_mindeg ← minimum degree column vertex in V;
5:     while inc > 0 do
6:         if deg(u_mindeg) ≤ deg(v_mindeg) then
7:             Find all column vertices adjacent to u_mindeg and decrease their
                   degree by 1;
8:             ListRow(u_mindeg) ← inc;
9:             inc − −;
10:            Assign next minimum degree row vertex as u_mindeg;
11:            Recompute v_mindeg;
12:        end if
13:        if deg(v_mindeg) ≤ deg(u_mindeg) then
14:            Find all row vertices adjacent to v_mindeg and decrease their de-
                   gree by 1;
15:            ListCol(v_mindeg) ← inc;
16:            inc − −;
17:            Recompute u_mindeg;
18:            Assign next minimum degree column vertex as v_mindeg;
19:        end if
20:    end while
21: end procedure
```

The combined ordering computed by SLO for matrix in Figure 4.1(a) is shown below.

ListRow

U	1	2	3	4	5
Ordering	**6**	**10**	**2**	**4**	**8**

ListCol

V	1	2	3	4	5
Ordering	**9**	**5**	**3**	**1**	**7**

The running time of smallest last ordering can be calculated as follows. The running time of steps at line numbers 3 and 4 is $O(m)$ and $O(n)$ respectively. The while statement on line 5 executes maximum of $(m+n)$ time in worst case. Inside the while loop, line 11 takes $O(n)$ time and line 17 takes $O(m)$ time while the remaining lines run for constant time. Thus the total running time of the while loop from statements in lines 5-20 is $O(\max\{m,n\}(m+n))$. Therefore, the running time of smallest last ordering algorithm is $O(\max\{m,n\}(m+n))$. Without loss of generality, let $m = \max\{m,n\}$, then the run time complexity of SLO algorithm is $O(m^2)$.

4.2.3 Incidence Degree Ordering

In incidence degree ordering (IDO) a row or column vertex w_k is determined after $w_1, w_2, ..., w_{k-2}, w_{k-1}$, where w_i is either a row vertex u_j or a column vertex v_l, have been selected. The choice of w_k from among the set of unordered vertices is such that it is adjacent to maximum number of already ordered vertices $\{w_1, w_2, ..., w_{k-2}, w_{k-1}\}$. The incidence degree of w_k is the degree of w_k in this subgraph.

In Algorithm 6, *inc* is the ordering counter which starts from 1. ListRow and ListCol, as described for LFO, store the row and column vertices already in the incidence degree and their ordering information. Lines 3, 4 find initial maximum degree row vertex u_{incdeg} and maximum degree column vertex v_{incdeg}. Statements in lines 5-11 initialize u_{incdeg} or v_{incdeg} as the first incidence degree vertex according to initial maximum degree amongst the two. The remaining incidence degree vertices and their orderings are computed in the statements of the while loop from lines 13-26. In line 14, degrees of all the column vertices

adjacent to already ordered row vertices are computed, and in line 15, degrees of all the row vertices adjacent to already ordered column vertices are computed. Lines 16 and 17 calculate u_{incdeg} and v_{incdeg}, i.e. the unordered row and column vertices that are adjacent to the maximum number of already ordered column and row vertices respectively. Statements in lines 18-24 find the new incidence degree row or column vertex and stores it in ListRow or ListCol with the ordering assigned to it.

The combined ordering computed by IDO for matrix in Figure 4.1(a) is shown below.

ListRow

U	1	2	3	4	5
Ordering	**1**	**5**	**9**	**6**	**3**

ListCol

V	1	2	3	4	5
Ordering	**2**	**4**	**10**	**7**	**8**

The running time of line 3 and 4 is $O(m)$ and $O(n)$ respectively. The while statement on 13 is executed $m+n$ times. Lines 14 and 17 are executed for maximum of n times each and lines 15 and 16 are executed for maximum of m times each, the remaining lines run for constant time. Thus the total running time of the while loop is $O(\max\{m,n\}(m+n))$, where $\max\{m,n\}$ denotes maximum of m,n. Therefore, the running time of incidence degree ordering algorithm is $O(\max\{m,n\}(m+n))$. Without loss of generality, let $m = \max\{m,n\}$ and, then the run time complexity of IDO algorithm is $O(m^2)$.

Algorithm 6 Incidence Degree Ordering

1: **procedure** IDO($G_b(A)$)
2: $inc \leftarrow 1$;
3: Find $u_{incdeg} \leftarrow$ maximum degree row vertex in U;
4: Find $v_{incdeg} \leftarrow$ maximum degree column vertex in V;
5: **if** $\deg(u_{incdeg}) \geq \deg(v_{incdeg})$ **then**
6: ListRow(u_{incdeg}) $\leftarrow inc$;
7: Remove u_{incdeg} from set of unordered vertices;
8: **else**
9: ListCol(v_{incdeg}) $\leftarrow inc$;
10: Remove v_{incdeg} from set of unordered vertices;
11: **end if**
12: $inc++$;
13: **while** inc $\neq (m+n)$ **do**
14: Find all unordered column vertices $v1_1, v1_2, ..., v1_n$ adjacent to vertices in ListRow and compute their incidence degrees;
15: Find all unordered row vertices $u1_1, u1_2, ..., u1_m$ adjacent to vertices in ListCol and compute their incidence degrees;
16: Find $u_{incdeg} \leftarrow$ maximum degree row vertex from $u1_1, ..., u1_m$;
17: Find $v_{incdeg} \leftarrow$ maximum degree column vertex from $v1_1, ..., v1_n$;
18: **if** $u_{incdeg} \geq v_{incdeg}$ **then**
19: ListRow(u_{incdeg}) $\leftarrow inc$;
20: Remove u_{incdeg} from set of unordered vertices;
21: **else**
22: ListCol(v_{incdeg}) $\leftarrow inc$;
23: Remove v_{incdeg} from set of unordered vertices;
24: **end if**
25: $inc++$;
26: **end while**
27: **end procedure**

In the following subsection we will describe the method to find bidirectional p-coloring using aforesaid ordering techniques.

4.2.4 Sequential Algorithm

After the vertices have been ordered using one of the ordering algorithms, the sequential algorithm will access the vertices in the given order and will assign the smallest available color to the vertices.

Algorithm 7, illustrates the sequential algorithm to assign colors to the vertices of $G_b(A)$. Variables *maxor* and *maxoc* represent the highest order number, in the combined ordering assigned to a row and a column vertex respectively. In lines 5 and 6 we construct two arrays Ordr and Ordc of size $m+n$ each to access the vertices corresponding to the combined ordering. To explain if the row and column vertices are ordered in the range $1, 2, ..., m+n$, then for each position $l \in \{1, 2, ..., m+n\}$ there can be exactly one vertex, either a row or a column, which is assigned the position l. This is implemented as Ordr(l) > 0, implying that the vertex in position l is a row vertex and consequently Ordc(l) is set to -1 indicating that there is no column vertex which is assigned position l in the combined ordering. Similarly, if Ordc(l) > 0 then Ordr(l) $= -1$. Finally, during the running of the algorithm Ordr(l) $= 0$ implies that the row vertex that was assigned order l has already been processed (colored).

The arrays Ordr and Ordc computed by Algorithm 7, using LFO for matrix in Figure 4.1(a) is shown below.

Ordr

Ordering	1	2	3	4	5	6	7	8	9	10
U	**1**	**4**	**-1**	**-1**	**2**	**3**	**5**	**-1**	**-1**	**-1**

Ordc

Ordering	1	2	3	4	5	6	7	8	9	10
V	-1	-1	**2**	**4**	-1	-1	-1	**1**	**3**	**5**

Let $\mathcal{C}$ be a group of columns. We say that $\mathcal{C}$ induces *direct determination* of the nonzero entries contained in those columns if for any j, k, l such that j, l are the indices of columns included in $\mathcal{C}$, we have $a_{kj} \neq 0$ and $a_{kl} \neq 0$, then there exists a row group $\mathcal{C}'$ from which the nonzero entries a_{kj} and a_{kl} have been determined.

(a) A (b) $G_b(A)$

Figure 4.2: Example to Illustrate Sequential Algorithm

In Algorithm 7, u_{minord} and v_{minord} are the vertices with minimum ordering among the ungrouped row and column vertices respectively. The while loop from 7-23 assigns the colors to the vertices. In line 8, we calculate the total number of nonzeros the row vertex u_{minord} can cover along with all other ungrouped row vertices which can be grouped with u_{minord}, such that if the group $\mathcal{C}'$ is formed, then it induces direct determination of the nonzero entries in the group. In line 9, we calculate the total number of nonzeros the column vertex v_{minord} can cover along with all other ungrouped column vertices which can be grouped with v_{minord}, such that if the group $\mathcal{C}$ is formed, then it induces direct determination of the nonzero entries in the group. In line 10 we check if

the number of nonzeros covered by vertex u_{minord} is more than the number of nonzeros covered by vertex v_{minord} and if this is the case then a new row group is formed otherwise a new column group is formed. We use tagging scheme as described by Gustavson [21] to form groups. Initially all the row or column vertices which can be part of one group are tagged and then the edges incident from these vertices are deleted. The process of forming groups or assigning colors continues until all the edges are accounted.

If we exit on line 20, and there still exist some row and column vertices which were not colored then on line 24 we assign the next available row color to all the remaining uncolored row vertices, similarly on line 25 we assign next available column color to all the remaining uncolored column vertices. The colors assigned on lines 24 and 25 are redundant colors, i.e. the nonzero entries in these color groups are already determined by other groups.

Proposition 4.1 *The sequential algorithm (Algorithm 7) computes a bidirectional coloring of* $G_b(A)$.

Proof. To show that the sequential algorithm produces a bidirectional coloring of the graph $G_b(A)$ we need to show that the vertices in every path of length 3 uses atleast three different colors. Now consider an arbitrary path $u_i - v_j - u_k - v_l$ as shown in Figure 4.2(b). Since the grouping of rows and columns as described in line 11 and 15 are such that the groups formed induce direct determination of the nonzeros and given the columns (or rows if it is a row group) in the first group are structurally orthogonal, we must have that either columns j and l are included in different column groups or the rows i and k are included in different row groups. Therefore, the total number of colors used on the vertices u_i, v_j, u_k, v_l are atleast three. Hence, the proof. □

Algorithm 7 Sequential Algorithm

```
1: procedure SEQUENTIAL(G_b(A))
2:     lor ← 1; loc ← 1;
3:     Find maxor;
4:     Find maxoc;
5:     Construct Array Ordr and calculate u_minord;
6:     Construct Array Ordc and calculate v_minord;
7:     while lor ≤ maxor && loc ≤ maxoc do
8:         Calculate the number of nonzeros covered by u_minord;
9:         Calculate the number of nonzeros covered by v_minord;
10:        if Number of nonzeros covered by vertex u_minord ≥ Number of
               nonzeros covered by vertex v_minord then
11:            Form a new row group;
12:            Delete edges in G_b(A) adjacent to the rows in this group;
13:            Set lor to the next minimum ordering number amongst the un-
                   grouped row vertices;
14:        else
15:            Form a new column group;
16:            Delete edges in G_b(A) adjacent to the columns in this group;
17:            Set loc to the next minimum ordering number amongst the un-
                   grouped column vertices;
18:        end if
19:        if G_b(A) contains no more edges then
20:            Exit from the while loop;
21:        end if
22:        Find next u_minord and v_minord;
23:    end while
24:    Assign next available row color to all the uncolored row vertices;
25:    Assign next available column color to all the uncolored column vertices;
26: end procedure
```

The running time of Algorithm 7, can be discussed as follows. The running time of lines 5 is $O(m)$ and lines 6 is $O(n)$. Since *maxor* or *maxoc* is equal to $m+n$ thus the while statement at line 7 executes $m+n$ times. Lines 11 and 12 runs m times each and lines 15 and 16 runs n times each. Rest of the lines takes constant time. Thus the total running time of the while loop 7-23 is $O(\max\{m,n\}(m+n))$, where $\max\{m,n\}$ denotes maximum of m,n. Therefore, the total running time of sequential algorithm is $O(\max\{m,n\}(m+n))$. Without loss of generality, let $m = \max\{m,n\}$, then the run time complexity of sequential algorithm is $O(m^2)$.

To check the validity of above algorithms, a validity check algorithm has been implemented that checks that groups formed follow the definition of bidirectional p-coloring as stated in section 3.2.2.

4.3 Summary

In this chapter we described unidirectional and bidirectional p-coloring techniques. We discussed existing heuristic algorithms for unidirectional and bidirectional p-coloring. We also described largest first ordering, smallest last ordering and, incidence degree ordering as adapted by us for bidirectional p-coloring of A. In chapter 6, we will show the experimental results of the heuristics and will have the comparative study of various coloring heuristics. In the next chapter we will explain exact coloring method for bidirectional p-coloring.

Chapter 5

Optimal Bidirectional Coloring

In this chapter we will describe exact bidirectional p-coloring techniques. In section 5.1 we will review the current literature. In section 5.2 we will discuss our exact coloring formulation together with complexity of the ILP model and implementation details. Finally, in section 5.3 we will conclude this chapter.

5.1 Background

Exact coloring refers to coloring the graph such that the number of colors assigned to the vertices of the graph is minimum and no better solution can be found. Finding this optimal solution is NP-hard [16]. In the following subsection we will discuss a selection of relevant optimal coloring algorithms.

5.1.1 DSATUR

DSATUR algorithm was developed by Brélaz [8] which is based on Randall-Brown's exact graph coloring algorithm [37]. DSATUR divides the graph coloring instance into a series of subproblems. A subproblem in DSATUR is a partial coloration of the graph. At each step there is an upper bound (UB) on the number of colors required to color the graph. If the subproblem uses p colors such that $p <$ UB, then a better coloring is found and UB is set to p. If the graph is not completely colored and the number of colors used is less than

UB, then new subproblems are created. An uncolored vertex v_i is chosen for branching and for each feasible color out of p colors a subproblem is created to assign that color to v_i. Another subproblem is created to assign color $p+1$ to v_i.

The choice of branch node i is critical and could affect the performance of the algorithm. Brélaz suggested to choose the node adjacent to the largest number of differently colored nodes. Sewell [39] suggested a modification to DSATUR noting that if the first p nodes colored form a clique, then these nodes would never be recolored. Thus it is useful to find a maximal clique in the graph and color those nodes first. This approach is a large improvement when the clique value and the coloring number of the graph are close.

Mehrotra and Trick [33] implemented the DSATUR algorithm by finding a large clique in the graph. The algorithm generates 10,000 clique subproblems and the rest of the nodes are dynamically ordered according to the number of adjacent colors and subproblems are created as in basic DSATUR. The subproblems are then solved in depth-first search manner to find the optimal coloring.

5.1.2 Branch and Cut Algorithm for Graph Coloring

Branch-and-cut methods [34] are exact algorithms consisting of a combination of a cutting plane method with a branch-and-bound algorithm. These methods solve a sequence of linear programming relaxations of the integer programming problem. Cutting plane methods improve the relaxation of the problem to closely approximate the integer programming problem, and branch-and-bound algorithms proceed by a sophisticated divide and conquer approach to solve problems.

Díaz and Zabala [15] proposed a branch-and-cut strategy to find optimal solution of general graph coloring problem. The problem is modelled with an

integer linear programming (ILP) formulation.

5.2 Exact Bidirectional Coloring

In this section we present the optimal bidirectional determination of Jacobian matrices using integer linear programming (ILP) method. The following subsection will discuss integer linear programming concept, followed by the presentation of the ILP model. Subsection thereafter will discuss the complexities of the model and the final subsection will present the implementation details.

5.2.1 Integer Linear Programming

A *linear programming* problem [31] is a mathematical program in which the objective function is linear in the unknowns and the constraints consists of linear equalities and linear inequalities. It can be expressed in the following standard form.

$$\begin{aligned} \text{minimize} \quad & c^T x \\ \text{subject to} \quad & Ax = b \\ & x \geq 0 \end{aligned}$$

where $x \in \Re^n$ is the vector of variables to be determined, $A \in \Re^{m \times n}$ is a matrix of known coefficients, and $c \in \Re^n$ and $b \in \Re^m$ are vectors of known coefficients. The expression $c^T x$ is called the objective function, and the equations $Ax = b$ are called the constraints. The variable x satisfying these constraints is said to be feasible for these constraints.

Integer linear programming (ILP) models [40] are the ones whose variables are constrained to take integers or whole numbers (as opposed to fractional values). The zero-one (or 0-1 or binary) variables restrict their integer variables to the values zero and one.

Integer programming is a much harder problem than ordinary linear programming problem. The problem of determining whether an ILP has an objective value less than a given target is a member of the class of "NP-complete" problems, all of which are very hard to solve. Since any NP-complete problem is reducible to any other, virtually any combinatorial problem of interest can be attacked in principle by solving some equivalent ILP.

Most available general-purpose large-scale ILP codes use "branch-and-bound" to search for an optimal integer solution by solving a sequence of related LP "relaxations" that allow some fractional values. It requires more computer time and memory to solve a ILP problem than to solve the corresponding LP relaxation. The difficulty of any particular ILP problem is hard to predict. Some problems with fewer variables can be challenging while other problems with larger number of variables can be solved readily. The best explanations of why a particular ILP is difficult often rely on some insight into the system to be modelled and it is observed that the way the model is formulated is as important as the actual choice of a solver.

5.2.2 Integer Linear Programming Model for Bidirectional p-coloring

We have attempted to find the optimal solution of bidirectional p-coloring for determining Jacobian matrices by developing an Integer Linear Programming (ILP) model. The detailed description of the model follows.

Let $A \in \Re^{m \times n}$ be a sparse matrix with known sparsity pattern and $G_b(A) = (U \cup V, E)$ the corresponding bipartite graph where U and V are the sets of vertices corresponding to the rows and columns of A respectively. We assume that the vertices in U are indexed $1, 2, ..., m$ and the vertices in V are indexed $m+1, m+2, ..., m+n$ and the quantities p_U and p_V denote upper bound on the number of colors we allow for the row and column vertices respectively. Below

is the description of binary variables (0-1) as used in the ILP formulation of bi-directional p-coloring.

- 0-1 variable w_j denotes whether ($w_j = 1$) or not ($w_j = 0$) color j, $1 \leq j \leq p_U$ has been assigned to some vertex $u \in U$.
- 0-1 variable w_j denotes whether ($w_j = 1$) or not ($w_j = 0$) color j, $p_U+1 \leq j \leq p_U + p_V$ has been assigned to some vertex $v \in V$.
- 0-1 variable $x_{i,j}$ denotes whether ($x_{i,j} = 1$) or not ($x_{i,j} = 0$) vertex i, $1 \leq i \leq m$ has been assigned color $j, 1 \leq j \leq p_U$.
- 0-1 variable $x_{i,j}$ denotes whether ($x_{i,j} = 1$) or not ($x_{i,j} = 0$) vertex i, $m+1 \leq i \leq m+n$ has been assigned color $j, p_U + 1 \leq j \leq p_U + p_V$.

ILP model for the computation of bidirectional chromatic number of $G_b(A)$ is as follows:

$$\text{minimize} \quad \sum_{j=1}^{p_U+p_V} w_j \tag{5.1}$$

$$\sum_{j=1}^{p_U} x_{i,j} = 1, \text{ for } i \in U \tag{5.2}$$

$$\sum_{j=p_U+1}^{p_U+p_V} x_{i,j} = 1, \text{ for } i \in V \tag{5.3}$$

$$x_{p,j} + x_{q,j'} + x_{r,j} + x_{s,j'} \leq (w_j + w_{j'} + 1)* \tag{5.4}$$

*for each path $p - q - r - s$ of length 3 and for each color pair $\{j, j'\}, 1 \leq j \leq p_U, p_U + 1 \leq j' \leq p_U + p_V$.

$$w_j \leq \sum_{i \in U} x_{i,j} \quad \text{for } j = 1, ..., p_U \tag{5.5}$$

$$w_j \leq \sum_{i \in V} x_{i,j} \quad \text{for } j = p_U + 1, ..., p_U + p_V \tag{5.6}$$

$$\sum_{i \in U} x_{i,j} \leq m w_j \quad \text{for } j = 1, ..., p_U \tag{5.7}$$

$$\sum_{i \in V} x_{i,j} \leq n w_j \quad \text{for } j = p_U + 1, ..., p_U + p_V \tag{5.8}$$

$$w_{j+1} \leq w_j \quad \text{for } j = 1, ..., p_U \tag{5.9}$$

$$w_{j+1} \leq w_j \quad \text{for } j = p_U + 1, ..., p_U + p_V \tag{5.10}$$

$$w_j \in \{0, 1\}, \quad \text{for } 1 \leq j \leq p_U + p_V \tag{5.11}$$

$$x_{i,j} \in \{0, 1\}, \quad \text{for } i \in U \cup V, 1 \leq j \leq p_U + p_V \tag{5.12}$$

Figure 5.1: ILP Formulation for Bidirectional p-coloring

Expression (5.1) represents the objective function to be minimized. Constraints (5.2) and (5.3) ensure that each vertex in the respective set of bipartition receives exactly one color. Constraint (5.4) enforce the coloring condition

for bidirectional p-coloring. Constraints (5.5) and (5.6) state that color w_j can only be greater than 0, if it has been assigned to a vertex. Constraints (5.7) and (5.8) ensure that the number of vertices assigned color j cannot be greater than the total number of vertices in the set U and V respectively. Constraint (5.9) and (5.10) ensure *minimal color assignment* to the vertices i.e. they ensure that the colors are assigned in ascending order of their ordering.

Proposition 5.1 *Any feasible solution of the bidirectional ILP induces a coloring of $G_b(A)$ such that each vertex in $G_b(A)$ receives exactly one color.*

Proof. We know that $x_{i,j}$ are the binary variables and thus can have value either 1 or 0. The sum $\sum_{j=1}^{p_U+p_V} x_{i,j}$ for $i \in U \cup V$ can be exactly 1 only if one of the variables $x_{i,j}$ has value 1. Constraints (5.2) and (5.3) ensures that in any feasible solution of above ILP model only one of $x_{i,j}$, $1 \leq j \leq p_U + p_V$ assumes the value of 1 for each $i \in U \cup V$. Consequently, vertex i receiving only 1 color. Thus by analogous reasoning it follows that each vertex in $U \cup V$ is assigned exactly one color via constraints (5.2) and (5.3). □

A path P is called *bi-colored* if the vertices forming P are colored with only two colors.

Proposition 5.2 *The bidirectional ILP has a feasible solution if and only if it induces a coloring ϕ of G_b in which no path of length 3 in G_b is bi-colored.*

Proof. We know that the vertices in U and V are assigned two disjoint sets of colors and any path of length 3 will have at least 2 colors. We will base the proof on the fact that $x_{i,j}$ can be assigned values either 1 or 0 in constraint (5.4) of the above ILP model.

Constraint (5.4) with path $p-q-r-s$ of length 3 where $p, q, r, s \in U \cup V$ and the color pair $\{j, j'\}$, $j \in \{1, ..., p_U\}$ and $j' \in \{p_U + 1, ..., p_U + p_V\}$ is associated with the feasible solution of bidirectional ILP. The following cases

will illustrate the possible ways to assign coloring to $x_{i,j}$ using constraint (5.4)

$$x_{p,j} + x_{q,j'} + x_{r,j} + x_{s,j'} \leq (w_j + w_{j'} + 1)$$

i $x_{p,j} = x_{r,j} = 1$ and $x_{q,j'} = x_{s,j'} = 1$ such that $\phi(p) = \phi(r) = j$ and $\phi(q) = \phi(s) = j'$. Then $x_{p,j} + x_{q,j'} + x_{r,j} + x_{s,j'} = 1 + 1 + 1 + 1 = 4$ while $w_j + w_{j'} = 1 + 1$ making the linear program infeasible and thus preventing bi-coloring of path P.

ii $x_{p,j} = x_{r,j} = 0$ and $x_{q,j'} = x_{s,j'} = 1$ such that $\phi(p) = \phi(r) = l \neq j$ and $\phi(q) = \phi(s) = j'$. Then $x_{p,j} + x_{q,j'} + x_{r,j} + x_{s,j'} = 0 + 1 + 0 + 1 = 2$ while

$$w_j + w_{j'} = \begin{cases} 1 + 1 = 2 \text{ if } x_{i,j} = 1 \text{ for some } i \in U \cup V \\ 0 + 1 = 1 \text{ if } x_{i,j} = 0 \text{ for all } i \in U \cup V \end{cases}$$

and hence satisfying the constraint (5.4) for $\{j, j'\}$ while bi-coloring the path P. But then the color pair $\{l, j'\}$ the inequality reduces to case (i) and hence making the solution infeasible as a result preventing bi-coloring of path P. Similarly, the assignment $x_{q,j'} = x_{s,j'} = 0$ and $x_{p,j} = x_{r,j} = 1$ such that $\phi(p) = \phi(r) = j$ and $\phi(q) = \phi(s) = l' \neq j$ corresponds to an infeasible solution too and thus disallowing bi-coloring of P.

iii $x_{p,j} = x_{r,j} = 0$ and $x_{q,j'} = x_{r,j'} = 0$ such that $\phi(p) = \phi(r) = l \neq j'$ and $\phi(q) = \phi(s) = l' \neq j'$. However, in this assignment of variables for the pair $\{l, l'\}$ results in the inequality which can be reduced to case (i), thus making the solution infeasible.

iv $x_{p,j} = 1, x_{r,j} = 0$ and $x_{q,j'} = x_{s,j'} = 1$ such that $\phi(p) = j, \phi(r) = l \neq j$ and $\phi(q) = \phi(s) = j'$. Then $x_{p,j} + x_{q,j'} + x_{r,j} + x_{s,j'} = 1 + 1 + 0 + 1 = 3$ and $w_j + w_{j'} = 1 + 1$ and hence satisfying constraint (5.4) for $\{j, j'\}$ while path P is colored using 3 different colors j, l, j'.

v $x_{p,j} = 0, x_{r,j} = 0$ and $x_{q,j'} = x_{s,j'} = 1$ such that $\phi(p) = k \neq j, \phi(r) = l \neq j$ and $\phi(q) = \phi(s) = j'$. Then $x_{p,j} + x_{q,j'} + x_{r,j} + x_{s,j'} = 0 + 1 + 0 + 1 = 2$

and

$$w_j + w_{j'} = \begin{cases} 1+1=2 \text{ if } x_{i,j} = 1 \text{ for some } i \in U \cup V \\ 0+1=1 \text{ if } x_{i,j} = 0 \text{ for all } i \in U \cup V \end{cases}$$

and hence satisfying constraint (5.4) while path P is colored using 3 different colors k, l, j'. This case is symmetric to case(ii).

vi $x_{p,j} = 0, x_{r,j} = 0$ and $x_{q,j'} = 1, x_{s,j'} = 0$ such that $\phi(p) = k \neq j, \phi(r) = l \neq j$ and $\phi(q) = j', \phi(s) = l' \neq j'$. Then $x_{p,j} + x_{q,j'} + x_{r,j} + x_{s,j'} = 0+1+0+0 = 2$ and

$$w_j + w_{j'} = \begin{cases} 1+1=2 \text{ if } x_{i,j} = 1 \text{ for some } i \in U \cup V \\ 0+1=1 \text{ if } x_{i,j} = 0 \text{ for all } i \in U \cup V \end{cases}$$

and hence satisfying constraint (5.4) while path P is colored using 4 different colors k, l, j', l'. This case is symmetric to case(ii).

vii $x_{p,j} = 0, x_{r,j} = 0$ and $x_{q,j'} = 0, x_{s,j'} = 0$ such that $\phi(p) = k \neq j, \phi(r) = l \neq j$ and $\phi(q) = k \neq j', \phi(s) = l' \neq j'$. Then $x_{p,j} + x_{q,j'} + x_{r,j} + x_{s,j'} = 0+0+0+0 = 0$ and

$$w_j + w_{j'} = \begin{cases} 1+1=2 & \text{if } x_{i,j} = 1 \text{ for some } i \in U \cup V \\ & \text{and } x_{i',j'} = 1 \text{ for some } i' \in U \cup V \\ 0+1=1 & \text{if } x_{i,j} = 0 \text{ for all } i \in U \cup V \\ & \text{and } x_{i',j'} = 1 \text{ for some } i' \in U \cup V \\ 1+0=1 & \text{if } x_{i,j} = 1 \text{ for some } i \in U \cup V \\ & \text{and } x_{i',j'} = 0 \text{ for all } i' \in U \cup V \\ 0+0=0 & \text{if } x_{i,j} = 0 \text{ for all } i \in U \cup V \\ & \text{and } x_{i',j'} = 0 \text{ for all } i' \in U \cup V \end{cases}$$

and hence satisfying constraint (5.4) while path P is colored using 4 different colors k, l, k', l'.

The above cases represents all the distinct assignments to variables $x_{i,j}$

associated with each path of length 3. In each case the infeasible solution corresponds to an invalid bidirectional p-coloring while a feasible solution corresponds to a valid bidirectional p-coloring. By proposition 1, a feasible solution induces a coloring of G_b where each vertex receives exactly one color. Hence this proves the proposition. □

We call color j *positive* if $w_j = 1$.

Proposition 5.3 *A vertex is assigned a color if and only if that color is positive*

Proof. Suppose color j with $1 \leq j \leq p_U$ is positive. Then $w_j = 1$. For inequality (5.5) to hold we must have some vertex $v_i, i \in U$ such that $x_{ij} = 1$. Since the color j can be used by at most $|U| = m$ vertices, constraint (5.7) also holds. With a similar reasoning for constraint (5.6) and (5.8) we can show the result for $i \in V$ with $p_U \leq j \leq p_U + p_V$.

Conversely, suppose color $j, 1 \leq j \leq p_U$, is not positive. Then $w_j = 0$. For inequality (5.7) to hold we must have that for all $1 \leq i \leq m$, $x_{ij} = 0$. With a similar reasoning for constraint (5.8) we can show the result for $i \in V$ with $p_U \leq j \leq p_U + p_V$. Hence the proof. □

Denote by $z_{\min}$ the value of the objective function in the optimum solution $\sigma_{\min}$ of the ILP of Figure 5.1.

Since a feasible solution to the ILP of Figure 5.1 induces a bidirectional p-coloring of $G_b(A)$ the following result is the direct consequence of the propositions 5.1, 5.2 and 5.3.

Theorem 5.4 *Given $A \in \Re^{m \times n}$, σ_{min} is the optimum solution of the ILP corresponding to $G_b(A)$ if and only if σ_{min} induces a bidirectional p-coloring of $G_b(A)$ such that $z_{min} = \chi(G_b(A))$.*

5.2.3 Complexities

In this section we will derive the computational complexity associated with the optimal bidirectional coloring. Following are the attributes related to our ILP model.

- Maximum number of variables for row color is p_U and for column color is p_V. Each row vertex can be assigned maximum of p_U colors. Thus for m rows maximum number of variables will be $m \times p_U$. Similarly, each column vertex can be assigned maximum of p_V colors. Thus for n columns maximum number of variables will be $n \times p_V$. Total number of variables in the ILP model are:

$$(n+1)p_V + (m+1)p_U \tag{5.13}$$

- Number of 3-paths:

$$\text{num3paths} = \sum_{i=1}^{m}(\rho_i - 1)\left[\sum_{j:a_{ij}\neq 0}(\kappa_j - 1)\right] \tag{5.14}$$

  ρ_i represents the number of nonzeros in row i and κ_j denotes the number of nonzeros in column j. Path of length 3 in a bipartite is denoted by "num3paths" and is of the order $O(\text{nnz}^2)$, where nnz are the number of nonzeros in the matrix.

- (5.2) have m constraints, (5.3) have n constraints, (5.4) have (num3paths$*$ $p_U * p_V$) constraints, (5.5) and (5.7) have p_U constraints each and (5.6) and (5.8) have p_V constraints each. (5.9) have $p_U - 1$ constraints and (5.10) have $p_V - 1$ constraint. Thus the total number of constraints are:

$$(\text{num3paths} * p_U * p_V) + (m+n) + 2(p_U + p_V) + (p_U + p_V - 2) \tag{5.15}$$

While solving a coloring problem, there are two kinds of symmetries [6, 36] that can be present in a solution. In the ILP model, the colors can be arbitrarily

permuted (instance-independent symmetries), and some graphs may remain unchanged under some permutations (instance-dependent symmetries). These symmetries affect the time and space complexities of the ILP model. One such kind of instance-independent symmetry occurring in our ILP model is discussed below.

Definition 5.5 *Null Color Symmetry* [36]. Consider a p-coloring problem with colors $1...p$ for a graph. Assuming that G can be optimally colored with $p-1$ colors, consider a solution where color i is not used:

$$(n_1, ..., n_{i-1}, (n_i =)0, n_{i+1}, ..., n_p),$$

where n_i denotes the number of vertices receiving color i. This assignment is equivalent to another assignment,

$$(n'_1, ..., n'_{j-1}, (n'_j =)0, n'_{j+1}, ..., n'_p),$$

where $i \neq j$ and $n_i = n'_j$. The color i for which $n_i = 0$ is called *null color*. For example, the assignment (1,0,2,3) is equivalent to (1,3,2,0), (0,1,2,3), (1,2,0,3). This is due to the existence of null colors, which create symmetries in an instance of p-coloring because any color can be swapped with a null color.

Constraints (5.9) and (5.10) deals with removing the null-color symmetries occurring in our ILP formulation.

Proposition 5.6 *The ILP in Figure 5.1 does not allow null colors.*

Proof. In a minimum coloring assignment, of the row vertices by constraint (5.9), color j can be positive only if color $j-1$ is positive and thus the colors not used in a solution automatically appear at the end of the coloring assignment and hence eliminating null colors. Similar argument can be applied to using constraint (5.10) for column vertices. Hence the proof. □

5.2.4 Implementation

In this section we will discuss the implementation details of the model described above.

Given a sparse matrix in Harwell-Boeing or Matrix Market format [2], we designed a program in C++ that generated the corresponding ILP instance for the bi-directional coloring of the associated graph. The generated ILP model was compatible with the CPLEX MIP solver [4]. A sample of the ILP model for a 2×2 arrowhead matrix is given in Appendix B.

5.3 Summary

In this chapter we described the optimal bidirectional p-coloring. We presented an ILP model for bidirectional p-coloring and discussed the complexities involved. We also looked into the implementation details of this model. We will present the experimental results for this implementation as well as for the heuristic bidirectional coloring in chapter 6.

Chapter 6

Experimental Results

In this chapter we will present computational results for coloring algorithms proposed in this thesis. For the purpose of comparison we also include unidirectional heuristic and exact coloring results. A more elaborate presentation of computational results is given in Appendix A. In section 6.1 we will provide the relevant features of the test problems. In section 6.2 we will give the heuristic and exact unidirectional coloring test results, followed by section 6.3 where we will give test results of various bidirectional heuristic techniques. In section 6.4 we will compare experimental results of heuristic and exact bidirectional coloring, followed by section 6.5 where we will compare results of unidirectional and bidirectional coloring. In section 6.6 we will summarize the coloring techniques for the determination of Jacobian matrices and finally in section 6.7 we will conclude the chapter.

6.1 Introduction

The details of the experimentation environment are as follows.

Machine: SUNW,Sun-Blade-100;sparc;sun4u

Operating system: SunOS Release 5.9 Generic_112233-12

Desktop: CDE 1.5.5, x11 Version 6.6.1

Physical memory (RAM): 256 Megabytes

Virtual memory (Swap): 681 Megabytes

For experimenting with our heuristic and exact techniques, matrices from Harwell-Boeing test matrices [1, 2, 3], and netlib library [5] were taken. Table 6.1 illustrates the properties of the matrices.

Table 6.1: Matrix Statistics

Matrix	n	m	nnz	DNSM	ρ_{max}	ρ_{min}	κ_{max}	κ_{min}
abb313	176	313	1557	2.83	6	1	26	2
adlittle	138	56	424	5.49	27	1	11	1
agg	615	488	2862	0.954	19	2	43	1
agg2	758	516	4740	1.21	49	2	43	1
agg3	758	516	4756	1.22	49	2	43	1
arc130	130	130	1282	7.59	124	1	124	1
ash219	85	219	438	2.35	2	2	9	2
ash292	292	292	2208	2.59	14	4	14	4
ash331	104	331	662	1.92	2	2	12	3
ash608	188	608	1216	1.06	2	2	12	2
ash958	292	958	1916	0.685	2	2	13	3
blend	114	74	522	6.19	29	2	16	1
bore3d	334	233	1448	1.86	73	1	28	1
bp0	822	822	3276	0.485	266	1	20	1
bp1000	822	822	4661	0.69	308	1	21	1
bp1200	822	822	4726	0.699	311	1	21	1
bp1400	822	822	4790	0.709	311	1	21	1
bp1600	822	822	4841	0.716	304	1	21	1
bp200	822	822	3802	0.563	283	1	21	1
bp400	822	822	4028	0.596	295	1	21	1
bp600	822	822	4172	0.617	302	1	21	1
bp800	822	822	4534	0.671	304	1	21	1
can1054	1054	1054	12196	1.1	35	6	35	6
can1072	1072	1072	12444	1.08	35	6	35	6
can256	256	256	2916	4.45	83	4	83	4
can268	268	268	3082	4.29	37	4	37	4
can292	292	292	2540	2.98	35	4	35	4
can634	634	634	7228	1.8	28	2	28	2
can715	715	715	6665	1.3	105	2	105	2
curtis54	54	54	291	9.98	12	3	16	3

Matrix	n	m	nnz	DNSM	ρ_{max}	ρ_{min}	κ_{max}	κ_{min}
dwt1007	1007	1007	8575	0.846	10	3	10	3
dwt1242	1242	1242	10426	0.676	12	2	12	2
dwt2680	2680	2680	25026	0.348	19	4	19	4
dwt419	419	419	3563	2.03	13	6	13	6
dwt59	59	59	267	7.67	6	2	6	2
eris1176	1176	1176	18552	1.34	99	2	99	2
fs541-1	541	541	4285	1.46	11	1	541	5
fs541-2	541	541	4285	1.46	11	1	541	5
gent113	113	113	655	5.13	20	1	27	1
ibm32	32	32	126	12.3	8	2	7	2
impcol-a	207	207	572	1.33	8	1	5	1
impcol-b	59	59	312	8.96	7	2	12	1
impcol-c	137	137	411	2.19	8	1	8	1
impcol-d	425	425	1339	0.741	10	1	10	1
impcol-e	225	225	1308	2.58	12	1	23	1
israel	316	174	2443	4.44	119	2	136	1
lundA	147	147	2449	11.3	21	5	21	5
lundB	147	147	2441	11.3	21	5	21	5
scagr25	671	471	1725	0.546	10	1	9	1
scagr7	185	129	465	1.95	10	1	9	1
shl0	663	663	1687	0.384	422	1	4	1
shl200	663	663	1726	0.393	440	1	4	1
shl400	663	663	1712	0.389	426	1	4	1
stair	614	356	4003	1.83	36	2	34	1
standata	1274	359	3230	0.706	745	2	10	1
str0	363	363	2454	1.86	34	1	34	1
str200	363	363	3068	2.33	30	1	26	1
str400	363	363	3157	2.4	33	1	34	1
str600	363	363	3279	2.49	33	1	34	1
tuff	628	333	4561	2.18	113	0	25	1
vtp-base	346	198	1051	1.53	38	1	12	1
watt2	1856	1856	11550	0.335	128	1	65	2
west0067	67	67	294	6.55	6	1	10	2
west0381	381	381	2157	1.49	25	1	50	1
west0497	497	497	1727	0.699	28	1	55	1
will199	199	199	701	1.77	6	1	9	2
will57	57	57	281	8.65	11	2	11	2

n - Number of columns in A

m - Number of rows in A

nnz - Number of nonzeros in A

DNSM - Matrix Density
ρ_{max} - Maximum number of nonzeros in any row
ρ_{min} - Minimum number of nonzeros in any row
κ_{max} - Maximum number of nonzeros in any column
κ_{min} - Minimum number of nonzeros in any column

6.2 Unidirectional Heuristic and Exact Coloring

In this section we will be presenting the computational test results of unidirectional heuristic and exact coloring. In unidirectional coloring a lower bound on the number of colors is the size of the largest clique in the graph as computed by DSM. The DSATUR [33] algorithm was implemented in C while DSM [9] was implemented in Fortran, and the running time of DSM was calculated using Perl code.

In table 6.2, column 2 depicts the lower bound found by DSM. Columns 3 and 4 give the number of colors and time taken by DSM and columns 5 and 6 illustrate the number of colors and time taken by DSATUR algorithm.

We observe that DSATUR algorithm is able to solve almost all the problems except fs541-1, fs541-2, dwt1007 and dwt2680. Leaving the above mentioned test problems, we find that the total of lower bound for all the matrices is 6429, the total number of colors for all matrices by DSM is 6444 and the total number of colors for all matrices by DSATUR algorithm is 6436. Thus we see that DSM is almost optimal.

The running time for both the algorithms is given in seconds. DSM takes 13 seconds to execute all the matrices while DSATUR takes 66.4 seconds. Since the algorithms were implemented in different languages. We cannot compare the running times of DSM and DSATUR accurately. However, roughly speaking we can say that the running times for the two algorithms are quite close to each other.

Table 6.2: DSM vs DSATUR

Matrix	Lower Bound	DSM	DSM Time	DSATUR	DSATUR Time
abb313	10	10	0	10	0.1
adlittle	27	27	0	27	0.0
agg	19	19	0	19	0.6
agg2	49	49	0	49	0.8
agg3	49	49	0	49	0.8
arc130	124	124	0	124	0.0
ash219	3	4	0	4	0.0
ash292	14	14	0	14	0.2
ash331	6	6	0	6	0.0
ash608	5	6	0	6	0.1
ash958	6	6	0	6	0.1
blend	29	29	1	29	0.0
bore3d	73	73	0	73	0.1
bp0	266	266	1	266	0.7
bp1000	308	308	1	308	0.8
bp1200	311	311	0	311	0.8
bp1400	311	311	0	311	0.8
bp1600	304	304	1	304	0.8
bp200	283	283	1	283	0.8
bp400	295	295	0	295	0.7
bp600	302	302	0	302	0.8
bp800	304	304	1	304	0.8
can1054	35	35	0	35	4.7
can1072	35	35	0	35	4.9
can256	83	83	0	83	0.1
can268	37	37	0	37	0.4
can292	35	35	0	35	0.1
can634	28	28	0	28	1.0
can715	105	105	0	105	0.6
curtis54	12	12	0	12	0.0
dwt1007	10	11	0	-	-
dwt1242	12	15	0	-	-
dwt2680	19	19	1	19	25.0
dwt419	14	15	0	15	6.0
dwt59	6	6	0	6	0.0
eris1176	99	99	1	99	1.8
fs541-1	11	13	0	-	-
fs541-2	11	13	0	-	-
gent113	20	20	0	20	0.0
ibm32	8	8	0	8	0.0

Matrix	Lower Bound	DSM	DSM Time	DSATUR	DSATUR Time
impcol-a	8	8	0	8	0.1
impcol-b	10	11	0	10	0.0
impcol-c	8	8	0	8	0.0
impcol-d	10	11	0	10	0.3
impcol-e	20	21	0	21	0.1
israel	119	119	0	119	0.1
lundA	21	22	0	21	0.1
lundB	21	24	0	21	0.1
scagr25	10	10	0	10	0.8
scagr7	10	10	0	10	0.1
shl0	422	422	0	422	0.5
shl200	440	440	0	440	0.5
shl400	426	426	0	426	0.5
stair	36	36	1	36	0.5
standata	745	745	1	745	1.9
str0	34	34	0	34	0.1
str200	30	30	0	30	0.1
str400	33	33	0	33	0.1
str600	33	33	0	33	0.2
tuff	113	114	0	114	1.0
vtp-base	38	38	0	38	0.1
watt2	128	128	1	128	4.2
west0067	7	9	1	8	0.0
west0381	27	29	1	28	0.2
west0497	28	28	0	28	0.3
will199	7	7	0	7	0.1
will57	11	11	0	11	0.0
Total	6429	6444	13	6436	66.4

- Represents that no result was found in 10 hours

6.3 Bidirectional Heuristics

In this section we present experimental test results of the heuristic techniques we implemented and compare these results with the existing bidirectional heuristics. Our heuristic algorithms were implemented in C++ on Sun Solaris Unix platform.

Table 6.3 compares our bidirectional heuristics results with complete direct cover [23] results. For each matrix we have taken the minimum of the number of colors obtained from LFO, SLO and IDO and this result is reported in the column named minLSI. Direct cover results are listed under the column named CDC. We find that for most of the matrices the number of colors are almost comparable. The total number of colors for all matrices in minLSI are 571(33) and total number of colors for all matrices in Direct Cover are 580(33). Also we notice that LFO results are more in agreement with that of complete direct cover as is expected since complete direct cover ordering is also based on the number of nonzeros (degrees) in rows and columns. The number inside the parentheses are the extra or redundant colors which were given to the vertices already covered by other colors. There could be at most two extra redundant colors, one for row and one for column vertices as described in section 4.2.4.

Table 6.3: Comparison of minLSI with Direct Cover Algorithm

Matrix	LFO	SLO	IDO	minLSI	CDC
abb313	13(1)	10(1)	10(1)	10(1)	13(1)
arc130	26(1)	131(1)	43(1)	26(1)	26(1)
ash219	5(1)	5(1)	5(1)	5(1)	5(1)
ash292	9(1)	8(1)	8(1)	8(1)	10(1)
ash331	6(1)	6(1)	6(1)	6(1)	6(1)
ash608	7(1)	6(1)	6(1)	6(1)	7(1)
ash958	7(1)	6(1)	6(1)	6(1)	6(1)
bp0	16(1)	20(1)	20(1)	16(1)	16(1)
bp1000	23(1)	25(1)	21(1)	21(1)	22(1)
bp1200	23(1)	21(1)	21(1)	21(1)	22(1)
bp1400	28(1)	21(1)	22(1)	21(1)	22(1)
bp1600	28(1)	21(1)	21(1)	21(1)	21(1)
bp200	17(1)	20(1)	21(1)	17(1)	18(1)
bp400	20(1)	21(1)	21(1)	20(1)	19(1)
bp600	22(1)	21(1)	21(1)	21(1)	18(1)
bp800	23(1)	22(1)	21(1)	21(1)	21(1)
curtis54	16(1)	16(1)	12(1)	12(1)	10(1)
eris1176	80(1)	81(1)	81(1)	80(1)	80(1)
fs541-1	16(1)	14(1)	15(1)	14(1)	15(1)

Matrix	LFO	SLO	IDO	minLSI	CDC
fs541-2	16(1)	14(1)	15(1)	14(1)	15(1)
gent113	19(1)	27(1)	24(1)	19(1)	18(1)
ibm32	8(1)	9(1)	8(1)	8(1)	8(1)
lundA	13(1)	13(1)	13(1)	13(1)	14(1)
lundB	15(1)	12(1)	13(1)	12(1)	14(1)
shl0	4(1)	4(1)	4(1)	4(1)	4(1)
shl200	4(1)	4(1)	4(1)	4(1)	4(1)
shl400	4(1)	4(1)	4(1)	4(1)	4(1)
str0	26(1)	26(1)	27(1)	26(1)	24(1)
str200	33(1)	30(1)	32(1)	30(1)	31(1)
str400	36(1)	33(1)	34(1)	33(1)	36(1)
str600	38(1)	33(1)	36(1)	33(1)	35(1)
will199	9(1)	8(1)	8(1)	8(1)	7(1)
will57	11(1)	11(1)	11(1)	11(1)	9(1)
Total				571(33)	580(33)

In Table 6.4 we compare our bidirectional heuristics results with bicoloring algorithm [12]. Again for each matrix we have taken the minimum of the number of colors obtained from LFO, SLO and IDO and this result is reported in column named minLSI. The results of bicoloring are reported in the column named Bi-col. For nearly all the considered matrices the results of minLSI and bicoloring are comparable except for israel, watt2 and west0497 where the results of minLSI are far better than that of bicoloring. The total number of colors for all matrices from minLSI is 595(29) and the total number of groups for bicoloring is 602.

Table 6.4: Comparison of minLSI with Bicoloring Algorithm

Matrix	LFO	SLO	IDO	minLSI	Bi-col
adlittle	11(1)	12(1)	12(1)	11(1)	11
agg	22(1)	20(1)	21(1)	20(1)	19
agg2	33(1)	31(1)	50(1)	31(1)	26
agg3	34(1)	29(1)	36(1)	29(1)	27
arc130	26(1)	131(1)	43(1)	26(1)	25
blend	20(1)	17(1)	22(1)	17(1)	16

Matrix	LFO	SLO	IDO	minLSI	Bi-col
bore3d	25(1)	28(1)	28(1)	25(1)	28
can1054	30(1)	38(1)	38(1)	30(1)	31
can1072	31(1)	36(1)	37(1)	31(1)	32
can256	29(1)	30(1)	56(1)	29(1)	32
can268	30(1)	40(1)	36(1)	30(1)	18
can292	19(1)	23(1)	37(1)	19(1)	17
can634	29(1)	29(1)	29(1)	29(1)	28
can715	21(1)	34(1)	27(1)	21(1)	22
gent113	19(1)	27(1)	24(1)	19(1)	19
impcol-c	6(1)	10(1)	9(1)	6(1)	6
impcol-d	6(1)	12(1)	12(1)	6(1)	6
impcol-e	22(1)	23(1)	23(1)	22(1)	21
israel	50(1)	55(1)	54(1)	50(1)	61
scagr25	8(1)	9(1)	9(1)	8(1)	8
scagr7	8(1)	9(1)	9(1)	8(1)	8
stair	38(1)	48(1)	36(1)	36(1)	36
standata	9(1)	10(1)	10(1)	9(1)	9
tuff	20(1)	26(2)	25(2)	20(1)	21
vtp-base	12(1)	16(1)	17(1)	12(1)	12
watt2	13(1)	65(1)	14(1)	13(1)	20
west0067	11(1)	11(1)	10(1)	10(1)	9
west0381	12(1)	12(1)	14(1)	12(1)	12
west0497	18(1)	16(1)	29(1)	16(1)	22
Total				595(29)	602

6.4 Heuristic and Exact Bidirectional

ILP instances were generated using Perl and C++ on Sun Solaris Unix platform. The generated ILP model was compatible with CPLEX MIP solver [4, 32] which was run under Windows XP Home Edition with AMD Athlon processor with 1GB RAM. Each problem was run for a maximum of 10 hours.

For small matrices the coloring results obtained are generally better than the heuristic coloring results. The current formulation of our ILP avoids null colors via a set of inequalities. By implementing null color symmetry breaking in our ILP model we have reduced the running time by approximately 3 folds.

Table 6.5 shows the results of minLSI and the ILP formulation. Due to time and memory constraints, we were able to get the results only for six matrices. We find that for 3 out of 6 matrices the number of colors found by exact ILP are fewer than the bidirectional heuristics. Also we see that the results of ash331, ash608 and impcol-a are same for both heuristic and bidirectional coloring and thereby optimal.

Table 6.5: Comparison of Heuristic and Exact Bidirectional Coloring

Matrix	minLSI			exact ILP		
	RG	*CG*	*TG*	*RG*	*CG*	*TG*
ibm32	1(1)	7(0)	8(1)	1(1)	6(0)	7(1)
ash219	0(1)	5(0)	5(1)	0(1)	4(0)	4(1)
ash331	0(1)	6(0)	6(1)	0(1)	6(0)	6(1)
ash608	0(1)	6(0)	6(1)	0(1)	6(0)	6(1)
impcol-a	5(1)	1(1)	6(2)	6(0)	0(1)	6(1)
impcol-c	1(1)	5(0)	6(1)	1(1)	3(0)	4(1)
Total			37(7)			33(6)

RG - Total number of row groups
CG - Total number of column groups
TG - RG + CG

6.5 Unidirectional and Bidirectional

In this section we compare the results of unidirectional and bidirectional heuristics. In Table 6.6 we see that for most of the matrices bidirectional techniques are far superior to unidirectional techniques with regard to the number of colors to completely determine the Jacobian matrices. Over 67 test problems, the total number of colors required by DSM and minLSI is 6496 and 1254(68) respectively. This is approximately a 5 fold reduction in the number of colors.

The total running time of all matrices for unidirectional matrices is 13 seconds while the total running time of all matrices for bidirectional heuristic is 19017 seconds.

Table 6.6: Comparison of Unidirectional and Bidirectional Coloring Heuristics

Matrix	DSM	DSM Time	minLSI	minLSI Time
abb313	10	0	10(1)	7
adlittle	27	0	11(1)	1
agg	19	0	20(1)	88
agg2	49	0	31(1)	137
agg3	49	0	29(1)	139
arc130	124	0	26(1)	2
ash219	4	0	5(1)	1
ash292	14	0	14(1)	13
ash331	6	0	6(1)	4
ash608	6	0	6(1)	25
ash958	6	0	6(1)	95
blend	29	1	17(1)	1
bore3d	73	0	25(1)	14
bp0	266	1	16(1)	309
bp1000	308	1	21(1)	283
bp1200	311	0	21(1)	282
bp1400	311	0	21(1)	286
bp1600	304	1	21(1)	288
bp200	283	1	17(1)	287
bp400	295	0	20(1)	286
bp600	302	0	21(1)	288
bp800	304	1	21(1)	282
can1054	35	0	30(1)	487
can1072	35	0	31(1)	512
can256	83	0	29(1)	8
can268	37	0	30(1)	10
can292	35	0	19(1)	12
can634	28	0	29(1)	114
can715	105	0	21(1)	146
curtis54	12	0	12(1)	0
dwt1007	11	0	11(1)	409
dwt1242	15	0	15(1)	772
dwt2680	19	1	21(1)	8419
dwt419	15	0	16(1)	34
dwt59	6	0	7(1)	0
eris1176	99	1	93(1)	732
fs541-1	13	0	14(1)	82
fs541-2	13	0	14(1)	84
gent113	20	0	19(1)	1
ibm32	8	0	8(1)	0

Matrix	DSM	DSM Time	minLSI	minLSI Time
impcol-a	8	0	6(2)	4
impcol-b	11	0	11(1)	0
impcol-c	8	0	6(1)	1
impcol-d	11	0	6(1)	35
impcol-e	21	0	22(1)	5
israel	119	0	50(1)	9
lundA	22	0	26(1)	1
lundB	24	0	26(1)	1
scagr25	10	0	8(1)	94
scagr7	10	0	8(1)	2
shl0	422	0	4(1)	177
shl200	440	0	4(1)	169
shl400	426	0	4(1)	175
stair	36	1	36(1)	55
standata	745	1	9(1)	250
str0	34	0	26(1)	26
str200	30	0	30(1)	24
str400	33	0	33(1)	27
str600	33	0	33(1)	26
tuff	114	0	20(1)	52
vtp-base	38	0	12(1)	11
watt2	128	1	13(1)	2840
west0067	9	1	10(1)	0
west0381	29	1	12(1)	28
west0497	28	0	16(1)	61
will199	7	0	8(1)	4
will57	11	0	11(1)	0
Total	6496	13	1254(68)	19017

Finally, in Table 6.7 we see that 4 out of 6 matrices have fewer number of colors in case bidirectional exact coloring as compared to unidirectional exact coloring. Notably, for the problem impcol-c we find that the number of colors required by bidirectional p-coloring is one-half of the number of colors required by unidirectional p-coloring.

Table 6.7: Comparison of Exact Unidirectional and Bidirectional Coloring

Matrix	DSATUR	ILP
ibm32	8	7(1)
ash219	4	4(1)
ash331	6	6(1)
ash608	6	6(1)
impcol-a	8	6(1)
impcol-c	8	4(1)
Total	40	33(6)

6.6 Final Results

Table 6.8 summarizes the results of Unidirectional heuristic and exact coloring, bidirectional heuristic results for LFO, SLO, IDO, and bidirectional exact coloring.

Table 6.8: Summary of all the Coloring Techniques

Matrix	DSM	DSATUR	LFO	SLO	IDO	Bi-Dir
abb313	10	10	13(1)	10(1)	10(1)	-
adlittle	27	27	11(1)	12(1)	12(1)	-
agg	19	19	22(1)	20(1)	21(1)	-
agg2	49	49	33(1)	31(1)	50(1)	-
agg3	49	49	34(1)	29(1)	36(1)	-
arc130	124	124	26(1)	131(1)	43(1)	-
ash219	4	4	5(1)	5(1)	5(1)	4(1)
ash292	14	14	15(1)	15(1)	14(1)	-
ash331	6	6	6(1)	6(1)	6(1)	6(1)
ash608	6	6	7(1)	6(1)	6(1)	6(1)
ash958	6	6	7(1)	6(1)	6(1)	-
blend	29	29	20(1)	17(1)	22(1)	-
bore3d	73	73	25(1)	28(1)	28(1)	-
bp0	266	266	16(1)	20(1)	20(1)	-
bp1000	308	308	23(1)	25(1)	21(1)	-
bp1200	311	311	23(1)	21(1)	21(1)	-
bp1400	311	311	28(1)	21(1)	22(1)	-
bp1600	304	304	28(1)	21(1)	21(1)	-
bp200	283	283	17(1)	20(1)	21(1)	-

Matrix	DSM	DSATUR	LFO	SLO	IDO	Bi-Dir
bp400	295	295	20(1)	21(1)	21(1)	-
bp600	302	302	22(1)	21(1)	21(1)	-
bp800	304	304	23(1)	22(1)	21(1)	-
can1054	35	35	30(1)	38(1)	38(1)	-
can1072	35	35	31(1)	36(1)	37(1)	-
can256	83	83	29(1)	30(1)	56(1)	-
can268	37	37	30(1)	40(1)	36(1)	-
can292	35	35	19(1)	23(1)	37(1)	-
can634	28	28	29(1)	29(1)	29(1)	-
can715	105	105	21(1)	34(1)	27(1)	-
curtis54	12	12	16(1)	16(1)	12(1)	-
dwt1007	11	-	11(1)	11(1)	11(1)	-
dwt1242	15	-	16(1)	15(1)	16(1)	-
dwt2680	19	19	22(1)	21(1)	21(1)	-
dwt419	15	15	16(1)	17(1)	19(1)	-
dwt59	6	6	8(1)	7(1)	7(1)	-
eris1176	99	99	93(1)	93(1)	100(1)	-
fs541-1	13	-	16(1)	14(1)	15(1)	-
fs541-2	13	-	16(1)	14(1)	15(1)	-
gent113	20	20	19(1)	27(1)	24(1)	-
ibm32	8	8	8(1)	9(1)	8(1)	7(1)
impcol-a	8	8	8(1)	6(2)	8(1)	6(1)
impcol-b	11	10	11(1)	11(1)	12(1)	-
impcol-c	8	8	6(1)	10(1)	9(1)	4(1)
impcol-d	11	10	6(1)	12(1)	12(1)	-
impcol-e	21	21	22(1)	23(1)	23(1)	-
israel	119	119	50(1)	55(1)	54(1)	-
lundA	22	21	26(1)	28(1)	28(1)	-
lundB	24	21	26(1)	26(1)	28(1)	-
scagr25	10	10	8(1)	9(1)	9(1)	-
scagr7	10	10	8(1)	9(1)	9(1)	-
shl0	422	422	4(1)	4(1)	4(1)	-
shl200	440	440	4(1)	4(1)	4(1)	-
shl400	426	426	4(1)	4(1)	4(1)	-
stair	36	36	38(1)	48(1)	36(1)	-
standata	745	745	9(1)	10(1)	10(1)	-
str0	34	34	26(1)	26(1)	27(1)	-
str200	30	30	33(1)	30(1)	32(1)	-
str400	33	33	36(1)	33(1)	34(1)	-
str600	33	33	38(1)	33(1)	36(1)	-
tuff	114	114	20(1)	26(2)	25(2)	-

Matrix	DSM	DSATUR	LFO	SLO	IDO	Bi-Dir
vtp-base	38	38	12(1)	16(1)	17(1)	-
watt2	128	128	13(1)	65(1)	14(1)	-
west0067	9	8	11(1)	11(1)	10(1)	-
west0381	29	28	12(1)	12(1)	14(1)	-
west0497	28	28	18(1)	16(1)	29(1)	-
will199	7	7	9(1)	8(1)	8(1)	-
will57	11	11	11(1)	11(1)	11(1)	-

- Represents that no result was found in 10 hours

6.7 Summary

In this chapter we presented the experimental results of unidirectional and bidirectional p-coloring. In most of the cases bidirectional techniques were found to be superior to the unidirectional techniques in terms of the number of colors needed to color the graph associated with the Jacobian matrix. For unidirectional coloring, the results of exact and heuristic methods are nearly the same. Also in case of unidirectional and bidirectional exact coloring method, exact bidirectional method needed fewer colors than the unidirectional exact method. On the basis of limited test results, we see that the heuristic bidirectional coloring results are not far from the exact bidirectional results. But this requires further investigation. In the next chapter we will conclude this thesis and give suggestions for future research.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

In this thesis we studied methods to determine sparse Jacobian matrices. We saw that by partitioning the Jacobian matrices, the sparsity information could be efficiently exploited. Two ways to partition the matrices were described namely unidirectional partitioning and bidirectional partitioning. We observed that the partitioning problem could be formulated as a graph coloring problem.

Unidirectional and bidirectional p-coloring techniques were described to color the vertices of column intersection graph and bipartite graph respectively such that the nonzero entries of the Jacobian matrices could be determined directly. We discussed the existing unidirectional exact and heuristic techniques and bidirectional heuristic techniques. We detailed our heuristic bidirectional p-coloring methods and proposed an exact ILP model for bidirectional determination. To the best of our knowledge this is the first attempt at using ILP techniques to solve the bidirectional determination of Jacobian matrices.

We tested the unidirectional and bidirectional p-coloring algorithms on selected problems from Harwell-Boeing test matrices [1, 2, 3] and netlib library [5]. We found that in most of the cases the bidirectional techniques did far better than the unidirectional methods. On the test problems considered our

bidirectional heuristic techniques require fewer (although not by a large margin) row and column groups than the complete direct cover [23] and bicoloring [12]. Our bidirectional p-coloring results were compared to the results obtained from exact ILP formulation. In 3 out of 6 cases the results were the same. However only a few of the ILP instances could be solved in the allotted time. Therefore, it is not quite clear how the bidirectional heuristics are performing in general. We note that while the bidirectional heuristics required more CPU time as compared with DSM, it is to be emphasized that the coloring step is done only once in an iterative scheme e.g. the Newton's method.

7.2 Future Research Directions

For future research on this work we would like to give the following suggestions.

- In case of bidirectional heuristic techniques we would like to improve the code such that the time taken by incidence degree ordering algorithm is decreased and in turn the overall running time is decreased.
- We would like to profile the code for bidirectional heuristic techniques by looking into variants of the ordering algorithms and by employing different tie-breaking strategies. We would also like to implement an efficient data structure so that the running time can be decreased further.
- Memory requirement in the ILP model can be improved by implementing heuristics such that the complete branch and bound tree is not stored while the CPLEX solver is searching for the solution. This can be done by changing the settings of the solver and experimenting accordingly.
- As evidenced by the computational tests, by removing the null color symmetry we were able to reduce the running time. Another idea to break symmetries existing in the model is by ordering [7] the colors.

Fixing colors of the clique vertices in the bipartite graph can also help in reducing symmetries. Both the ideas could result in a reduction of running time.

- We would like to perform more elaborate numerical testing for exact ILP bidirectional p-coloring.

Bibliography

[1] ftp://ftp.cerfacs.fr/pub/algo/matrices/harwell_boeing/ (2005).

[2] http://math.nist.gov/matrixmarket/collections/hb.html (2005).

[3] http://math.nist.gov/matrixmarket/matrices.html (2005).

[4] http://www.ilog.com/products/cplex/ (2005).

[5] http://www.netlib.org/lp/data/ (2005).

[6] F.A. Aloul. Solving difficult SAT instances in the presence of symmetry. In *IEEE Trans. on CAD*, volume 22, pages 1117–1137. 2003.

[7] Rob H. Bisseling, Jaroslaw Byrka, Selin Cerav-Erbas, Nebojsa Gvozdenović, Mathias Lorenz, Rudi Pendavingh, Colin Reeves, Matthias Röger, and Arie Verhoeven. Partitioning a call graph. Technical report, Universiteit Utrecht, June 2005.

[8] Daniel Brélaz. New methods to color the vertices of a graph. *Commun. ACM*, 22(4):251–256, 1979.

[9] Thomas F. Coleman, Burton S. Garbow, and Jorge J. Moré. Software for estimating sparse Jacobian matrices. *ACM Trans. Math. Softw.*, 10(3):329–345, 1984.

[10] Thomas F. Coleman and Jorge J. Moré. Estimation of sparse Jacobian matrices and graph coloring problems. *SIAM J. Numer. Anal.*, 20(1):187–209, 1983.

[11] Thomas F. Coleman and Arun Verma. Structure and efficient Jacobian calculation. Technical report, Computer Science Department, Cornell University, 1996.

[12] Thomas F. Coleman and Arun Verma. The efficient computation of sparse Jacobian matrices using automatic differentiation. *SIAM J. Sci. Comput.*, 19(4):1210–1233, July 1998.

[13] A. R. Curtis, M. J. D. Powell, and J. K. Reid. On the estimation of sparse Jacobian matrices. *J. Inst. Math. Appl.*, 13:117–119, 1974.

[14] J. E. Dennis and Robert B. Schnabel. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations.* Prentice-Hall, Englewood Cliffs, NJ, 1983.

[15] Isabel Méndez Díaz and Paual Zabala. A branch-and-cut algorithm for graph coloring. Technical report, Universidad de Buenos Aires - Argentina, 2002.

[16] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., New York, NY, USA, 1990.

[17] Assefaw Hadish Gebremedhin, Fredrik Manne, and Alex Pothen. Graph coloring in optimization revisited. Technical report, Department of Informatics, University of Bergen, 2003.

[18] Assefaw Hadish Gebremedhin, Fredrik Manne, and Alex Pothen. What color is your Jacobian? Graph coloring for computing derivatives. Technical report, Accepted by SIAM Review, 2004.

[19] Andreas Griewank. *Evaluating derivatives: principles and techniques of algorithmic differentiation.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

[20] Rajiv Gupta, Mary Lou Soffa, and Denise Ombres. Efficient register allocation via coloring using clique separators. *ACM Trans. Program. Lang. Syst.*, 16(3):370–386, 1994.

[21] F. G. Gustavson. *Sparse Matrix Computations*, chapter Finding the block lower triangular form of a sparse matrix, pages 275–289. Academic Press, New York, 1976.

[22] A. K. M. Shahadat Hossain. *On The Computation of Sparse Jacobian Matrices and Newton Steps.* PhD thesis, Department of Informatics, University of Bergen, Norway, 1997.

[23] A. K. M. Shahadat Hossain and Trond Steihaug. Computing a sparse Jacobian matrix by rows and columns. *Optimization Methods and Software*, 10:33–48, 1998.

[24] Shahadat Hossain and Trond Steihaug. Reducing the number of AD passes for computing a sparse Jacobian matrix. In *Automatic Differentiation of Algorithms: From Simulation to Optimization*, Computer and Information Science, chapter 31, pages 263–270. Springer, New York, NY, 2001.

[25] Shahadat Hossain and Trond Steihaug. Sparsity issues in the computation of Jacobian matrices. In *ISSAC '02: Proceedings of the 2002 international symposium on Symbolic and algebraic computation*, pages 123–130, New York, NY, USA, 2002. ACM Press.

[26] Shahadat Hossain and Trond Steihaug. Optimal direct determination of sparse Jacobian matrices. Technical Report 254, University of Lethbridge and University of Bergen, October 2003.

[27] Shahadat Hossain and Trond Steihaug. Graph coloring in the estimation of sparse derivative matrices: Instances and applications, 2004. University of Lethbridge, Canada.

[28] Shahadat Hossain and Zhenshuan Zhang. CsegGraph: Column segment graph generator, 2003. University of Lethbridge, Canada.

[29] Sven O. Krumke, Madhav V. Marathe, and S. S. Ravi. Models and approximation algorithms for channel assignment in radio networks. *Wirel. Netw.*, 7(6):575–584, 2001.

[30] Yaw-Ling Lin and Steven S. Skiena. Algorithms for square roots of graphs. *SIAM J. Discret. Math.*, 8(1):99–118, 1995.

[31] David G. Luenberger. *Introduction to linear and nonlinear programming.* Addison-Wesley Publishing Company, 1973.

[32] Irvin Lustig. Embedding cplex using the ILOG CPLEX callable library. Technical report, http://optimization.ilog.com.

[33] Anuj Mehrotra and Michael A. Trick. A column generation approach for graph coloring. *INFORMS Journal on computing*, 8(4):344–354, 1996.

[34] John E. Mitchell. Branch-and-cut algorithms for combinatorial optimization problems. In *Notebook of Applied Optimization,*. Oxford University Press, 2000.

[35] G. N. NewSam and J. D. Ramsdell. Estimation of sparse Jacobian matrices. In *SIAM J. Alg. Disc. Meth.*, volume 4, pages 404–417. 1983.

[36] Arathi Ramani, Fadi A. Aloul, Igor L. Markov, and Karem A. Sakallah. Breaking instance-independent symmetries in exact graph coloring. In *Design Automation and Test Conference in Europe*, pages 324–329, February 2004.

[37] J Randall-Brown. Chromatic scheduling and the chromatic number problems. In *Management Science*, volume 4 of *Part I*, pages 456–463. December 1972.

[38] Timothy Anton Redl. *A Study of University Timetabling that Blends Graph Coloring with the Satisfaction of Various Essential and Preferential Conditions.* PhD thesis, Rice University, 2004.

[39] Edward C. Sewell. An improved algorithm for exact graph coloring. In *DIMACS Series on Discrete Mathematics and Theoretical Computer Science.* 1995.

[40] Robert J. Vanderbei. *Linear Programming, foundations and extensions.* Kluwer Academic Publishers, 2003.

Appendix A

Extended Heuristic Bidirectional Coloring Results

This appendix gives complete results of heuristic bidirectional coloring techniques.

Table A.1: LFO Result

Matrix	n	m	nnz	DNSM	ρ_{max}	ρ_{min}	κ_{max}	κ_{min}	CG	RG	TG
abb313	176	313	1557	2.83	6	1	26	2	12(0)	1(1)	13(1)
adlittle	138	56	424	5.49	27	1	11	1	0(1)	11(0)	11(1)
agg	615	488	2862	0.954	19	2	43	1	22(0)	0(1)	22(1)
agg2	758	516	4740	1.21	49	2	43	1	31(0)	2(1)	33(1)
agg3	758	516	4756	1.22	49	2	43	1	32(0)	2(1)	34(1)
arc130	130	130	1282	7.59	124	1	124	1	16(1)	10(0)	26(1)
ash219	85	219	438	2.35	2	2	9	2	5(0)	0(1)	5(1)
ash292	292	292	2208	2.59	14	4	14	4	0(1)	15(0)	15(1)
ash331	104	331	662	1.92	2	2	12	3	6(0)	0(1)	6(1)
ash608	188	608	1216	1.06	2	2	12	2	7(0)	0(1)	7(1)
ash958	292	958	1916	0.685	2	2	13	3	7(0)	0(1)	7(1)
blend	114	74	522	6.19	29	2	16	1	14(0)	6(1)	20(1)
bore3d	334	233	1448	1.86	73	1	28	1	3(1)	22(0)	25(1)
bp0	822	822	3276	0.485	266	1	20	1	2(1)	14(0)	16(1)
bp1000	822	822	4661	0.69	308	1	21	1	19(0)	4(1)	23(1)
bp1200	822	822	4726	0.699	311	1	21	1	18(1)	5(0)	23(1)
bp1400	822	822	4790	0.709	311	1	21	1	23(1)	5(0)	28(1)
bp1600	822	822	4841	0.716	304	1	21	1	15(1)	13(0)	28(1)
bp200	822	822	3802	0.563	283	1	21	1	7(0)	10(1)	17(1)
bp400	822	822	4028	0.596	295	1	21	1	15(1)	5(0)	20(1)
bp600	822	822	4172	0.617	302	1	21	1	13(1)	9(0)	22(1)
bp800	822	822	4534	0.671	304	1	21	1	20(0)	3(1)	23(1)
can1054	1054	1054	12196	1.1	35	6	35	6	8(1)	22(0)	30(1)
can1072	1072	1072	12444	1.08	35	6	35	6	9(1)	22(0)	31(1)
can256	256	256	2916	4.45	83	4	83	4	8(1)	21(0)	29(1)
can268	268	268	3082	4.29	37	4	37	4	7(1)	23(0)	30(1)
can292	292	292	2540	2.98	35	4	35	4	3(1)	16(0)	19(1)
can634	634	634	7228	1.8	28	2	28	2	0(1)	29(0)	29(1)
can715	715	715	6665	1.3	105	2	105	2	1(1)	20(0)	21(1)
curtis54	54	54	291	9.98	12	3	16	3	0(1)	16(0)	16(1)
dwt1007	1007	1007	8575	0.846	10	3	10	3	0(1)	11(0)	11(1)
dwt1242	1242	1242	10426	0.676	12	2	12	2	0(1)	16(0)	16(1)
dwt2680	2680	2680	25026	0.348	19	4	19	4	0(1)	22(0)	22(1)
dwt419	419	419	3563	2.03	13	6	13	6	0(1)	16(0)	16(1)
dwt59	59	59	267	7.67	6	2	6	2	0(1)	8(0)	8(1)
eris1176	1176	1176	18552	1.34	99	2	99	2	85(0)	8(1)	93(1)
fs541-1	541	541	4285	1.46	11	1	541	5	13(0)	3(1)	16(1)
fs541-2	541	541	4285	1.46	11	1	541	5	13(0)	3(1)	16(1)
gent113	113	113	655	5.13	20	1	27	1	16(0)	3(1)	19(1)
ibm32	32	32	126	12.3	8	2	7	2	7(0)	1(1)	8(1)

Matrix	n	m	nnz	DNSM	ρ_{max}	ρ_{min}	κ_{max}	κ_{min}	CG	RG	TG
impcol-a	207	207	572	1.33	8	1	5	1	8(0)	0(1)	8(1)
impcol-b	59	59	312	8.96	7	2	12	1	10(0)	1(1)	11(1)
impcol-c	137	137	411	2.19	8	1	8	1	5(0)	1(1)	6(1)
impcol-d	425	425	1339	0.741	10	1	10	1	5(0)	1(1)	6(1)
impcol-e	225	225	1308	2.58	12	1	23	1	20(0)	2(1)	22(1)
israel	316	174	2443	4.44	119	2	136	1	11(1)	39(0)	50(1)
lundA	147	147	2449	11.3	21	5	21	5	0(1)	26(0)	26(1)
lundB	147	147	2441	11.3	21	5	21	5	0(1)	26(0)	26(1)
scagr25	671	471	1725	0.546	10	1	9	1	3(0)	5(1)	8(1)
scagr7	185	129	465	1.95	10	1	9	1	3(0)	5(1)	8(1)
shl0	663	663	1687	0.384	422	1	4	1	0(1)	4(0)	4(1)
shl200	663	663	1726	0.393	440	1	4	1	0(1)	4(0)	4(1)
shl400	663	663	1712	0.389	426	1	4	1	0(1)	4(0)	4(1)
stair	614	356	4003	1.83	36	2	34	1	26(0)	12(1)	38(1)
standata	1274	359	3230	0.706	745	2	10	1	1(1)	8(0)	9(1)
str0	363	363	2454	1.86	34	1	34	1	18(1)	8(0)	26(1)
str200	363	363	3068	2.33	30	1	26	1	26(0)	7(1)	33(1)
str400	363	363	3157	2.4	33	1	34	1	32(0)	4(1)	36(1)
str600	363	363	3279	2.49	33	1	34	1	31(0)	7(1)	38(1)
tuff	628	333	4561	2.18	113	0	25	1	4(1)	16(0)	20(1)
vtp-base	346	198	1051	1.53	38	1	12	1	7(1)	5(0)	12(1)
watt2	1856	1856	11550	0.335	128	1	65	2	1(1)	12(0)	13(1)
west0067	67	67	294	6.55	6	1	10	2	3(1)	8(0)	11(1)
west0381	381	381	2157	1.49	25	1	50	1	4(1)	8(0)	12(1)
west0497	497	497	1727	0.699	28	1	55	1	8(1)	10(0)	18(1)
will199	199	199	701	1.77	6	1	9	2	9(0)	0(1)	9(1)
will57	57	57	281	8.65	11	2	11	2	1(1)	10(0)	11(1)

n - Number of columns in A

m - Number of rows in A

nnz - Number of nonzeros in A

DNSM - Matrix Density

ρ_{max} - Maximum number of nonzeros in any row

ρ_{min} - Minimum number of nonzeros in any row

κ_{max} - Maximum number of nonzeros in any column

κ_{min} - Minimum number of nonzeros in any column

RG - Total number of row groups

CG - Total number of column groups

TG - RG + CG

Table A.2: SLO Result

Matrix	n	m	nnz	DNSM	ρ_{max}	ρ_{min}	κ_{max}	κ_{min}	CG	RG	TG
abb313	176	313	1557	2.83	6	1	26	2	10(0)	0(1)	10(1)
adlittle	138	56	424	5.49	27	1	11	1	0(1)	12(0)	12(1)
agg	615	488	2862	0.954	19	2	43	1	20(0)	0(1)	20(1)
agg2	758	516	4740	1.21	49	2	43	1	25(0)	6(1)	31(1)
agg3	758	516	4756	1.22	49	2	43	1	25(0)	4(1)	29(1)
arc130	130	130	1282	7.59	124	1	124	1	124(0)	7(1)	131(1)
ash219	85	219	438	2.35	2	2	9	2	5(0)	0(1)	5(1)
ash292	292	292	2208	2.59	14	4	14	4	15(0)	0(1)	15(1)
ash331	104	331	662	1.92	2	2	12	3	6(0)	0(1)	6(1)
ash608	188	608	1216	1.06	2	2	12	2	6(0)	0(1)	6(1)
ash958	292	958	1916	0.685	2	2	13	3	6(0)	0(1)	6(1)
blend	114	74	522	6.19	29	2	16	1	0(1)	17(0)	17(1)
bore3d	334	233	1448	1.86	73	1	28	1	0(1)	28(0)	28(1)
bp0	822	822	3276	0.485	266	1	20	1	0(1)	20(0)	20(1)
bp1000	822	822	4661	0.69	308	1	21	1	2(1)	23(0)	25(1)
bp1200	822	822	4726	0.699	311	1	21	1	0(1)	21(0)	21(1)
bp1400	822	822	4790	0.709	311	1	21	1	0(1)	21(0)	21(1)
bp1600	822	822	4841	0.716	304	1	21	1	0(1)	21(0)	21(1)
bp200	822	822	3802	0.563	283	1	21	1	6(1)	14(0)	20(1)
bp400	822	822	4028	0.596	295	1	21	1	8(0)	13(1)	21(1)
bp600	822	822	4172	0.617	302	1	21	1	0(1)	21(0)	21(1)
bp800	822	822	4534	0.671	304	1	21	1	1(1)	21(0)	22(1)
can1054	1054	1054	12196	1.1	35	6	35	6	2(1)	36(0)	38(1)
can1072	1072	1072	12444	1.08	35	6	35	6	36(0)	0(1)	36(1)
can256	256	256	2916	4.45	83	4	83	4	26(0)	4(1)	30(1)
can268	268	268	3082	4.29	37	4	37	4	40(0)	0(1)	40(1)
can292	292	292	2540	2.98	35	4	35	4	5(1)	18(0)	23(1)
can634	634	634	7228	1.8	28	2	28	2	29(0)	0(1)	29(1)
can715	715	715	6665	1.3	105	2	105	2	12(1)	22(0)	34(1)
curtis54	54	54	291	9.98	12	3	16	3	0(1)	16(0)	16(1)
dwt1007	1007	1007	8575	0.846	10	3	10	3	0(1)	11(0)	11(1)
dwt1242	1242	1242	10426	0.676	12	2	12	2	15(0)	0(1)	15(1)
dwt2680	2680	2680	25026	0.348	19	4	19	4	21(0)	0(1)	21(1)
dwt419	419	419	3563	2.03	13	6	13	6	17(0)	0(1)	17(1)
dwt59	59	59	267	7.67	6	2	6	2	7(0)	0(1)	7(1)
eris1176	1176	1176	18552	1.34	99	2	99	2	87(0)	6(1)	93(1)
fs541-1	541	541	4285	1.46	11	1	541	5	2(1)	12(0)	14(1)
fs541-2	541	541	4285	1.46	11	1	541	5	2(1)	12(0)	14(1)
gent113	113	113	655	5.13	20	1	27	1	0(1)	27(0)	27(1)
ibm32	32	32	126	12.3	8	2	7	2	8(0)	1(1)	9(1)

Matrix	n	m	nnz	DNSM	ρ_{max}	ρ_{min}	κ_{max}	κ_{min}	CG	RG	TG
impcol-a	207	207	572	1.33	8	1	5	1	1(1)	5(1)	6(2)
impcol-b	59	59	312	8.96	7	2	12	1	10(0)	1(1)	11(1)
impcol-c	137	137	411	2.19	8	1	8	1	8(0)	2(1)	10(1)
impcol-d	425	425	1339	0.741	10	1	10	1	11(0)	1(1)	12(1)
impcol-e	225	225	1308	2.58	12	1	23	1	20(0)	3(1)	23(1)
israel	316	174	2443	4.44	119	2	136	1	20(1)	35(0)	55(1)
lundA	147	147	2449	11.3	21	5	21	5	27(0)	1(1)	28(1)
lundB	147	147	2441	11.3	21	5	21	5	25(0)	1(1)	26(1)
scagr25	671	471	1725	0.546	10	1	9	1	0(1)	9(0)	9(1)
scagr7	185	129	465	1.95	10	1	9	1	0(1)	9(0)	9(1)
shl0	663	663	1687	0.384	422	1	4	1	0(1)	4(0)	4(1)
shl200	663	663	1726	0.393	440	1	4	1	0(1)	4(0)	4(1)
shl400	663	663	1712	0.389	426	1	4	1	0(1)	4(0)	4(1)
stair	614	356	4003	1.83	36	2	34	1	32(0)	16(1)	48(1)
standata	1274	359	3230	0.706	745	2	10	1	0(1)	10(0)	10(1)
str0	363	363	2454	1.86	34	1	34	1	19(0)	7(1)	26(1)
str200	363	363	3068	2.33	30	1	26	1	30(0)	0(1)	30(1)
str400	363	363	3157	2.4	33	1	34	1	33(0)	0(1)	33(1)
str600	363	363	3279	2.49	33	1	34	1	33(0)	0(1)	33(1)
tuff	628	333	4561	2.18	113	0	25	1	1(1)	25(1)	26(2)
vtp-base	346	198	1051	1.53	38	1	12	1	4(1)	12(0)	16(1)
watt2	1856	1856	11550	0.335	128	1	65	2	0(1)	65(0)	65(1)
west0067	67	67	294	6.55	6	1	10	2	9(0)	2(1)	11(1)
west0381	381	381	2157	1.49	25	1	50	1	4(1)	8(0)	12(1)
west0497	497	497	1727	0.699	28	1	55	1	9(0)	7(1)	16(1)
will199	199	199	701	1.77	6	1	9	2	8(0)	0(1)	8(1)
will57	57	57	281	8.65	11	2	11	2	11(0)	0(1)	11(1)

n - Number of columns in A

m - Number of rows in A

nnz - Number of nonzeros in A

DNSM - Matrix Density

ρ_{max} - Maximum number of nonzeros in any row

ρ_{min} - Minimum number of nonzeros in any row

κ_{max} - Maximum number of nonzeros in any column

κ_{min} - Minimum number of nonzeros in any column

RG - Total number of row groups

CG - Total number of column groups

TG - RG + CG

Table A.3: IDO Result

Matrix	n	m	nnz	DNSM	ρ_{max}	ρ_{min}	κ_{max}	κ_{min}	CG	RG	TG
abb313	176	313	1557	2.83	6	1	26	2	10(0)	0(1)	10(1)
adlittle	138	56	424	5.49	27	1	11	1	0(1)	12(0)	12(1)
agg	615	488	2862	0.954	19	2	43	1	21(0)	0(1)	21(1)
agg2	758	516	4740	1.21	49	2	43	1	48(0)	2(1)	50(1)
agg3	758	516	4756	1.22	49	2	43	1	33(0)	3(1)	36(1)
arc130	130	130	1282	7.59	124	1	124	1	35(0)	8(1)	43(1)
ash219	85	219	438	2.35	2	2	9	2	5(0)	0(1)	5(1)
ash292	292	292	2208	2.59	14	4	14	4	0(1)	14(0)	14(1)
ash331	104	331	662	1.92	2	2	12	3	6(0)	0(1)	6(1)
ash608	188	608	1216	1.06	2	2	12	2	6(0)	0(1)	6(1)
ash958	292	958	1916	0.685	2	2	13	3	6(0)	0(1)	6(1)
blend	114	74	522	6.19	29	2	16	1	21(0)	1(1)	22(1)
bore3d	334	233	1448	1.86	73	1	28	1	0(1)	28(0)	28(1)
bp0	822	822	3276	0.485	266	1	20	1	0(1)	20(0)	20(1)
bp1000	822	822	4661	0.69	308	1	21	1	0(1)	21(0)	21(1)
bp1200	822	822	4726	0.699	311	1	21	1	0(1)	21(0)	21(1)
bp1400	822	822	4790	0.709	311	1	21	1	0(1)	22(0)	22(1)
bp1600	822	822	4841	0.716	304	1	21	1	0(1)	21(0)	21(1)
bp200	822	822	3802	0.563	283	1	21	1	0(1)	21(0)	21(1)
bp400	822	822	4028	0.596	295	1	21	1	0(1)	21(0)	21(1)
bp600	822	822	4172	0.617	302	1	21	1	0(1)	21(0)	21(1)
bp800	822	822	4534	0.671	304	1	21	1	0(1)	21(0)	21(1)
can1054	1054	1054	12196	1.1	35	6	35	6	3(1)	35(0)	38(1)
can1072	1072	1072	12444	1.08	35	6	35	6	1(1)	36(0)	37(1)
can256	256	256	2916	4.45	83	4	83	4	53(0)	3(1)	56(1)
can268	268	268	3082	4.29	37	4	37	4	34(0)	2(1)	36(1)
can292	292	292	2540	2.98	35	4	35	4	2(1)	35(0)	37(1)
can634	634	634	7228	1.8	28	2	28	2	0(1)	29(0)	29(1)
can715	715	715	6665	1.3	105	2	105	2	22(0)	5(1)	27(1)
curtis54	54	54	291	9.98	12	3	16	3	12(0)	0(1)	12(1)
dwt1007	1007	1007	8575	0.846	10	3	10	3	11(0)	0(1)	11(1)
dwt1242	1242	1242	10426	0.676	12	2	12	2	0(1)	16(0)	16(1)
dwt2680	2680	2680	25026	0.348	19	4	19	4	21(0)	0(1)	21(1)
dwt419	419	419	3563	2.03	13	6	13	6	0(1)	19(0)	19(1)
dwt59	59	59	267	7.67	6	2	6	2	7(0)	0(1)	7(1)
eris1176	1176	1176	18552	1.34	99	2	99	2	1(1)	99(0)	100(1)
fs541-1	541	541	4285	1.46	11	1	541	5	1(1)	14(0)	15(1)
fs541-2	541	541	4285	1.46	11	1	541	5	1(1)	14(0)	15(1)
gent113	113	113	655	5.13	20	1	27	1	17(0)	7(1)	24(1)
ibm32	32	32	126	12.3	8	2	7	2	7(0)	1(1)	8(1)

Matrix	n	m	nnz	DNSM	ρ_{max}	ρ_{min}	κ_{max}	κ_{min}	CG	RG	TG
impcol-a	207	207	572	1.33	8	1	5	1	8(0)	0(1)	8(1)
impcol-b	59	59	312	8.96	7	2	12	1	11(0)	1(1)	12(1)
impcol-c	137	137	411	2.19	8	1	8	1	9(0)	0(1)	9(1)
impcol-d	425	425	1339	0.741	10	1	10	1	10(0)	2(1)	12(1)
impcol-e	225	225	1308	2.58	12	1	23	1	20(0)	3(1)	23(1)
israel	316	174	2443	4.44	119	2	136	1	19(1)	35(0)	54(1)
lundA	147	147	2449	11.3	21	5	21	5	0(1)	28(0)	28(1)
lundB	147	147	2441	11.3	21	5	21	5	0(1)	28(0)	28(1)
scagr25	671	471	1725	0.546	10	1	9	1	0(1)	9(0)	9(1)
scagr7	185	129	465	1.95	10	1	9	1	0(1)	9(0)	9(1)
shl0	663	663	1687	0.384	422	1	4	1	0(1)	4(0)	4(1)
shl200	663	663	1726	0.393	440	1	4	1	0(1)	4(0)	4(1)
shl400	663	663	1712	0.389	426	1	4	1	0(1)	4(0)	4(1)
stair	614	356	4003	1.83	36	2	34	1	0(1)	36(0)	36(1)
standata	1274	359	3230	0.706	745	2	10	1	0(1)	10(0)	10(1)
str0	363	363	2454	1.86	34	1	34	1	20(0)	7(1)	27(1)
str200	363	363	3068	2.33	30	1	26	1	29(1)	3(0)	32(1)
str400	363	363	3157	2.4	33	1	34	1	33(0)	1(1)	34(1)
str600	363	363	3279	2.49	33	1	34	1	33(0)	3(1)	36(1)
tuff	628	333	4561	2.18	113	0	25	1	0(1)	25(1)	25(2)
vtp-base	346	198	1051	1.53	38	1	12	1	4(1)	13(0)	17(1)
watt2	1856	1856	11550	0.335	128	1	65	2	1(1)	13(0)	14(1)
west0067	67	67	294	6.55	6	1	10	2	9(0)	1(1)	10(1)
west0381	381	381	2157	1.49	25	1	50	1	4(1)	10(0)	14(1)
west0497	497	497	1727	0.699	28	1	55	1	28(0)	1(1)	29(1)
will199	199	199	701	1.77	6	1	9	2	8(0)	0(1)	8(1)
will57	57	57	281	8.65	11	2	11	2	0(1)	11(0)	11(1)

n - Number of columns in A

m - Number of rows in A

nnz - Number of nonzeros in A

DNSM - Matrix Density

ρ_{max} - Maximum number of nonzeros in any row

ρ_{min} - Minimum number of nonzeros in any row

κ_{max} - Maximum number of nonzeros in any column

κ_{min} - Minimum number of nonzeros in any column

RG - Total number of row groups

CG - Total number of column groups

TG - RG + CG

Appendix B

Example of ILP Model Implementation

A sample of the ILP model for a 2 × 2 arrowhead matrix is given below.

```
//Model File
range boolean 0..1;
enum rown ...;
enum coln ...;
enum rowc ...;
enum colc ...;
//Decision Variables
var boolean xr[rown,rowc];
var boolean xc[coln,colc];
var boolean wr[rowc];
var boolean wc[colc];
//Objective Function
minimize
    sum(r in rowc) wr[r] + sum(c in colc) wc[c]
//Constraints
subject to (
```

```
    forall(r in rown) sum(row in rowc) xr[r,row] = 1;
    forall(c in coln) sum(col in colc) xc[c,col] = 1;
    forall(r in rowc, c in colc) (
    xr[r0,r] + xc[c0,c] + xr[r1,r] + xc[c1,c] <= wr[r] + wc[c] + 1;
    xr[r0,r] + xc[c1,c] + xr[r1,r] + xc[c0,c] <= wr[r] + wc[c] + 1;
    xr[r1,r] + xc[c0,c] + xr[r0,r] + xc[c1,c] <= wr[r] + wc[c] + 1;
    xr[r1,r] + xc[c1,c] + xr[r0,r] + xc[c0,c] <= wr[r] + wc[c] + 1;
    );
    forall(r in rowc) wr[r] <= sum(row in rown) xr[row,r];
    forall(c in colc) wc[c] <= sum(col in coln) xc[col,c];
    wr[rc0] >= wr[rc1];
    wc[cc0] >= wc[cc1];
    forall(r in rowc) sum(row in rown) xr[row,r] <= 2*wr[r];
    forall(c in colc) sum(col in coln) xc[col,c] <= 2*wc[c];
);
//Data File
rown = (r0,r1);
coln = (c0,c1);
rowc = (rc0,rc1);
colc = (cc0,cc1);
```