BI-DIRECTIONAL DETERMINATION OF SPARSE JACOBIAN MATRICES: ALGORITHMS AND LOWER BOUNDS

ANIK SAHA
Bachelor of Science,Chittagong University of Engineering & Technology, 2008

A Thesis
Submitted to the School of Graduate Studies
of the University of Lethbridge
in Partial Fulfillment of the
Requirements for the Degree

MASTER OF SCIENCE

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

BI-DIRECTIONAL DETERMINATION OF SPARSE JACOBIAN MATRICES: ALGORITHMS AND LOWER BOUNDS

ANIK SAHA

Date of Defense: August 25, 2015

Dr. Shahadat Hossain Supervisor	Associate Professor	Ph.D.
Dr. Daya Gaur Committee Member	Professor	Ph.D.
Dr. Robert Benkoczi Committee Member	Associate Professor	Ph.D.
Dr. Christopher Anand McMaster University, Ontario External Examiner	Associate Professor	Ph.D.
Dr. Hadi Kharaghani Chair, Thesis Examination Committee	Professor	Ph.D.

Abstract

Efficient estimation of large sparse Jacobian matrices is a requisite in many large-scale scientific and engineering problems. It is known that estimation of non-zeroes of a matrix can be viewed as a graph coloring problem. Due to the presence of dense rows or dense columns, unidirectional partitioning does not always give good results. Bi-Directional partitioning handles the problem of dense rows and dense columns quite well[16].

Lower bound to determine the non-zeroes of a sparse Jacobian matrix can be defined as the least number of groups necessary to determine the matrix. For unidirectional partitioning, a good lower bound is given by the maximum number of non-zeroes in any row[4]. For bi-directional determination, both columns and rows must be considered to obtain a lower bound. In this thesis, we provide an easily computed better lower bound.

We have developed a heuristic algorithm and an iterative algorithm to determine non-zeroes of sparse Jacobian matrices using Bi-Directional partitioning. Our heuristic algorithm is inspired from graph coloring problems and recursive largest first partitioning of graphs. Our algorithm provides better result than the existing algorithms. For the iterative method, we have introduced randomization technique to color the vertices of the graph.

A part of our work was presented at "Applied Mathematics, Modelling and Computational Science" (AMMCS-2015).

Acknowledgments

At first, I would like to give my heartiest gratitude to my M.Sc. Supervisor Dr. Shahadat Hossain for his continuous support, proper guidance, helpful suggestions, and persistent encouragement throughout the period of my Masters Degree.

I would also like to express my sincere appreciation, obligation and indebtedness to Professor Dr. Daya Gaur for his co-operation, encouragement, attention to details and guidance. I am very grateful to him for his valuable comments and discussions. I would also like to express to Dr. Robert Benkoczi for taking the time to review my thesis. His valuable feedback has fed me in the successful completion of my thesis.

I am also grateful to my friends, colleagues and family for their inspiration and support.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Minimizing a non-linear function of large number of variables or finding numerical solution of a system of non-linear equations is often needed in order to solve complex scientific and engineering problems. Estimation of first or higher order derivatives of a vector function of several independent variables is required in many algorithms. The calculation of first order partial derivatives i.e. the Jacobian matrix is an important step in Newton's method. For large size problems, the Jacobian matrix is often sparse. Exploiting the sparsity pattern is often needed while calculating sparse Jacobian matrices. If we can divide the non-zero entries of the sparse Jacobian matrix into a group of rows and columns in such away that no two rows from the same group have non-zero entries in the same column and no two columns from the same group have non-zero entries in the same row, then the computation of the matrix becomes more efficient. For brevity, we will call the non-zero entries in matrices as "non-zeroes" in the remaining part of the thesis. Non-zeroes in each column group can be determined from finite difference approximation or forward mode of automatic differentiation and non-zeroes in the row groups can be determined by reverse mode of automatic differentiation [13]. Therefore the cost to determine a Jacobian matrix can be approximated by the cost of one forward calculation of automatic differentiation multiplied by the number of column groups plus the cost of one reverse calculation of automatic differentiation multiplied by the number of row groups. Hence the smaller the number of groups, the faster the computation of the Jacobian matrix.

1.1 Jacobian matrix

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

$$J(x) = F'(x) = \begin{pmatrix} \frac{\partial}{\partial x_1} f_1(x) & \frac{\partial}{\partial x_2} f_1(x) & \cdots & \frac{\partial}{\partial x_n} f_1(x) \\ \frac{\partial}{\partial x_1} f_2(x) & \frac{\partial}{\partial x_2} f_2(x) & \cdots & \frac{\partial}{\partial x_n} f_2(x) \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial}{\partial x_1} f_m(x) & \frac{\partial}{\partial x_2} f_m(x) & \cdots & \frac{\partial}{\partial x_n} f_m(x) \end{pmatrix} \tag{1.1}$$

1.2 Newton's method

In numerical analysis, Newton's method is an iterative method for finding the root of a real valued function $F(x) = 0$, where $F = (f_1, f_2, \ldots f_m)^T$ is a mapping $F : \Re^n \to \Re^m$. The steps of Newton's method are as follows:

NEWTON-METHOD()
1 Let $x \in \Re^n$ be given
2 **while** *notconverged*
3 Evaluate $b = F(x)$
4 Determine $J = F'(x)$
5 Solve $Js = -b$ for s
6 Update $x = x + s$

Figure 1.1: Newton's method to solve non-linear equations.

Let us interpret the algorithm of Newton's method with an example. Consider a vector function F as defined below

$$F(x) = \begin{bmatrix} x_1 + x_2 - 5 \\ x_1^2 - x_2^2 - 5 \end{bmatrix} \tag{1.2}$$

with root $\begin{bmatrix} 3 & 2 \end{bmatrix}^T$. The first order derivative of the matrix is as follows

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

Let $x = \begin{bmatrix} 0 & 4 \end{bmatrix}^T$. For the first iteration in Figure 1.1, $b = \begin{bmatrix} -1 & -21 \end{bmatrix}^T$ and

$$J = \begin{bmatrix} 1 & 1 \\ 0 & -8 \end{bmatrix}$$

The value of s is then found by solving

$$\begin{aligned} Js &= -b \\ \implies s = -J^{-1}b &= -\begin{bmatrix} 1 & 1 \\ 0 & -8 \end{bmatrix}^{-1} \begin{bmatrix} -1 \\ -21 \end{bmatrix} \\ \implies s &= \begin{bmatrix} 3.625 \\ -2.625 \end{bmatrix} \end{aligned}$$

Hence new value of x will be $\begin{bmatrix} 3.625 & 1.375 \end{bmatrix}^T$.

For the second iteration, $b = \begin{bmatrix} 3.625 + 1.375 - 5 \\ 3.625^2 - 1.375^2 - 5 \end{bmatrix} = \begin{bmatrix} 0 \\ 6.25 \end{bmatrix}$.

The value of J will be

$$J = \begin{bmatrix} 1 & 1 \\ 7.25 & -2.75 \end{bmatrix}$$

Hence $s = -\begin{bmatrix} 1 & 1 \\ 7.25 & -2.75 \end{bmatrix}^{-1} \begin{bmatrix} 0 \\ 6.25 \end{bmatrix} = \begin{bmatrix} -0.625 \\ 0.625 \end{bmatrix}$

Therefore new value of x will be $\begin{bmatrix} 3 & 2 \end{bmatrix}^T$. From the example, we can see that after two iterations, we found the original root of the function. If our assumption is far away from the root then more iterations are needed to get the roots of the equation. If we examine the algorithm, then we can see that in each iteration we need to evaluate $F(x)$ and its first order derivative $F'(x)$ at a given point x. So the computation of the first order partial derivatives is an essential step in finding solutions of a system of non-linear equations.

1.3 Finite difference approximation

The Jacobian matrix can be obtained by approximating it using a finite difference formula. For example, if $F : \Re^n \to \Re^m$ is a continuously differentiable mapping, then the j-th column of the Jacobian matrix at x can be obtained from

$$J_j(x) = \frac{\partial}{\partial x_j} F(x) \approx \frac{1}{\xi}[F(x+\xi e_j) - F(x)], \;\; 1 \leq j \leq n \tag{1.3}$$

where e_j is the j-th co-ordinate vector and ξ is a positive increment. If the function $F(x)$ has already been evaluated, then the j-th column of the matrix J can be approximated through the additional evaluation of $F(x+\xi e_j)$.

Let us consider a function

$$F(x) = \begin{bmatrix} x_1 + x_2 - 5 \\ x_1^2 - x_2^2 - 5 \end{bmatrix} \tag{1.4}$$

which is continuously differentiable. At some point $x = a$ where $a^T = \begin{bmatrix} 1.0 & 2.0 \end{bmatrix}$, the value of $F(x)$ will be

$$F(x) = F(a) = \begin{bmatrix} -2 \\ -8 \end{bmatrix}$$

The first order derivative of the matrix i.e. the Jacobian matrix is

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

At $x = a$, we get our Jacobian matrix as

$$J = F'(x) = \begin{bmatrix} 1 & 1 \\ 2 & -4 \end{bmatrix}$$

The two unit co-ordinate vectors needed for this calculation are $e_1^T = \begin{bmatrix} 1 & 0 \end{bmatrix}$ and $e_2^T = \begin{bmatrix} 0 & 1 \end{bmatrix}$. Let us assume the value ξ is 0.1

Now, for e_1 we get $F'(x)|_{x=a} \approx \begin{bmatrix} 1 \\ 2.1 \end{bmatrix}$ and for e_2, $F'(x)|_{x=a} \approx \begin{bmatrix} 1 \\ -4.1 \end{bmatrix}$

Hence using the finite difference approximation, the first order derivative of $F(x)$ at the point $x = a$ is $J(x) = F'(x)|_{x=a} \approx \begin{bmatrix} 1 & 1 \\ 2.1 & -4.1 \end{bmatrix}$, contains an error proportional to the step size $\xi = 0.1$.

Finite difference approximation is easy to implement, but if ξ is too large, then approximation may not be accurate due to truncation error. Also if ξ is too small, then the accuracy is compromised due to rounding errors. Truncation error is the error that is caused by neglecting the higher order terms in mathematical series and approximating it to a finite sum. Rounding error is caused by performing arithmetic operations in fixed precisions.

1.4 Automatic differentiation

Automatic differentiation is a chain rule based technique that is used to compute derivatives of a function with respect to the given arguments without incurring truncation error. Automatic differentiation manipulates an algorithmic specifications of a function and produces derivative information at selected arguments. Consider the function $f = \begin{bmatrix} f_1 & f_2 \end{bmatrix}^T$ defined as follows:

$$\begin{aligned} f_1(x_1,x_2) &= x_1 + x_2 - 5 \\ f_2(x_1,x_2) &= x_1^2 - x_2^2 - 5 \end{aligned} \tag{1.5}$$

The steps involved in computation of f is given as a sequence of arithmetic operations

$$\begin{aligned} v_1 &= x_1 \\ v_2 &= x_2 \\ v_3 &= v_1 + v_2 \\ v_4 &= v_1^2 \\ v_5 &= v_2^2 \\ v_6 &= v_4 - v_5 \\ v_7 &= v_3 - 5 \\ v_8 &= v_6 - 5 \end{aligned} \tag{1.6}$$

where v_i, $i = 3, 4, \ldots, 6$ are intermediate quantities. If we have the values of x_1 and x_2 then the result of the computation is obtained in

$$\begin{aligned} f_1(x_1,x_2) &= v_7 \\ f_2(x_1,x_2) &= v_8 \end{aligned} \tag{1.7}$$

The sequence of operations in Equation 1.6 is known as the code-list. It is possible to form different code-lists for the same function. After forming a code-list, we can apply rules of differentiation to compute derivative of a function with respect to the independent variables x_1 and x_2. Let $\nabla_k = \begin{pmatrix} \frac{\partial}{\partial x_1} v_k \\ \frac{\partial}{\partial x_2} v_k \end{pmatrix}$ denotes the gradient of v_k, $k = 1,2,\ldots,8$ with respect to independent variables, then $\nabla v_1 = \begin{bmatrix} 1 & 0 \end{bmatrix}^T$, $\nabla v_2 = \begin{bmatrix} 0 & 1 \end{bmatrix}^T$. For $k = 3,4,\ldots,8$

we have $v_k = \phi_k(v_i, v_j)$ where $i, j < k$. By the chain rule

$$\nabla_k = \frac{\partial}{\partial v_i}\phi_k \nabla v_i + \frac{\partial}{\partial v_j}\phi_k \nabla v_j$$

If the elementary partial derivatives $\frac{\partial}{\partial v_i}\phi_k$, $k = 3,4,\ldots,6$ can be computed, the code-list in Equation 1.6 can be enhanced to compute the derivatives

$$\begin{aligned}
v_1 &= x_1, & \nabla v_1^T &= (1,0) \\
v_2 &= x_2, & \nabla v_2^T &= (0,1) \\
v_3 &= v_1 + v_2, & \nabla v_3^T &= \nabla v_1^T + \nabla v_2^T, & yields(1,1) \\
v_4 &= v_1^2, & \nabla v_4^T &= 2v_1 \nabla v_1^T, & yields(2v_1,0) \\
v_5 &= v_2^2, & \nabla v_5^T &= 2v_2 \nabla v_2^T, & yields(0,2v_2) \\
v_6 &= v_4 - v_5, & \nabla v_6^T &= \nabla v_4^T - \nabla v_5^T, & yields(2v_1,-2v_2) \\
v_7 &= v_3 - 5, & \nabla v_7^T &= \nabla v_3^T, & yields(1,1) \\
v_8 &= v_6 - 5, & \nabla v_8^T &= \nabla v_6^T, & yields(2v_1,-2v_2)
\end{aligned} \tag{1.8}$$

For each calculated code-list (Equation 1.8), the corresponding gradient is also obtained. The final results of the computation are the function values of the Jacobian matrix

$$J(x_1,x_2) = \begin{pmatrix} \nabla v_7^T \\ \nabla v_8^T \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 2x_1 & -2x_2 \end{pmatrix} \tag{1.9}$$

1.4.1 Forward mode and reverse mode

The matrix representation that we got from the accumulation of elementary partial derivatives, captures two basic modes of Automatic differentiation, forward mode and reverse mode. If all these elementary functions ϕ_j are well defined and have continuous partial derivatives

$$c_{ji} \equiv \frac{\partial}{\partial v_i}\phi_j, \;\; i < j \tag{1.10}$$

then by repeated application of the chain rule, the non-zeroes of the Jacobian matrix $J(x)$ can be computed from the elementary partial derivatives c_{ji} .

Forward mode involves accumulation of intermediate partial derivative in the same order as the function values are computed. In forward mode, a forward pass involves calculation of matrix vector product Jv, where v is a n-vector. Then all columns of J can be computed by n forward passes. While in reverse mode, accumulation of intermediate partial derivatives are computed in reverse order as the function values are computed. A reverse pass requires the computation of $w^T J$, where w is a m-vector and all the rows of J can be determined by m reverse passes. For an extensive treatment of Automatic Differentiation we refer to [13].

1.5 Objective

The main objective of this thesis is to design and implement efficient methods to determine non-zeroes of large sparse Jacobian matrices. For large-scale problems, the Jacobian computation may dominate overall computation for complicated functions. When problem dimension is large and underlying Jacobian matrix is sparse, it is desirable to utilize the sparsity property to improve the efficiency of the first order partial derivative computation. If the sparsity pattern of the Jacobian matrix is known a priori and it does not change from iteration to iteration, then an approximation of the Jacobian matrix can be obtained by finite difference approximation or by employing automatic differentiation techniques.

Given the sparsity structure of a matrix A, we want to obtain vectors d_1, d_2, ..., d_p so that the elements of the given matrix A are uniquely determined from the products Ad_1, Ad_2, ..., Ad_p with p as small as possible. We can achieve it by dividing the non-zeroes into group of columns in such away that, no two columns from the same group have non-zeroes in the same row position. Then, the non-zeroes in each group can be determined by one finite difference application or by the application of the forward mode of automatic differentiation. The computation cost to determine the matrix is the cost of computing one finite

difference approximation or one application of forward mode of automatic differentiation multiplied by the number of column groups in the partition. But, if there are dense rows in the sparse matrix, then the method of partitioning columns does not always give good result. Partitioning rows into groups where no two rows from same group have non-zeroes in the same column position exploits these type of sparsity effectively. Using the reverse mode of automatic differentiation, it is possible to exploit the sparsity in rows. Partitioning the rows can also be stated as to find vectors d_1, d_2, ... , d_q so that the non-zeroes of the matrix A are uniquely determined from the products $A^T d_1$, $A^T d_2$, ..., $A^T d_q$ with q as small as possible[16]. But if the matrix has both dense rows and dense columns, then using row partition or column partition exclusively may not be able to exploit sparsity. In this case bi-directional partitioning, i.e. row partitioning and column partitioning together is preferable. For the rest of the thesis, A will be used to represent a sparse Jacobian matrix, m and n will be used to represent as the number of rows and the number of columns of A, r_i will be used to represent the i-th row where $i = 1, 2, \ldots, m$, c_j will be used to represent the j-th column where $j = 1, 2, \ldots, n$, a_{ij} will be used to represent a non-zero in the i-th row and j-th column, $nnz(A)$ will be used to represent total number of non-zeroes in A.

1.6 Our contribution

In this thesis, we have developed algorithms to obtain good lower bounds on the number of groups required to determine the sparse Jacobian matrix. We have also implemented two algorithms: one using a heuristic approach and the other using an iterative approach to find the minimum number of groups, a combination of row and column groups such that the non-zeroes of the matrix can be uniquely and directly determined. Our heuristic approach produces better results compared to the methods proposed in [11],[14],[21]. In our iterative approach, we introduced a randomization technique to partition the rows and columns of the matrix. To the best of our knowledge this is the first work that uses randomization to compute the sparse Jacobinan matrix. Including this chapter there are six more chapters in

this thesis.

In Chapter 2, we review basic graph concepts used in this thesis. We also describe an efficient data structure to store a sparse matrix in computer memory for our algorithms. Graph coloring methods that we have used are also discussed in this chapter.

In Chapter 3, we provide algorithms that calculate lower bounds on the number of groups in a bi-directional partitioning. We calculate lower bound of each square sub-matrix. The largest lower bound value among the sub-matrices is a lower bound on the number of groups in a bi-directional determination of the sparse Jacobian matrix.

In Chapter 4, we describe our heuristic algorithm to group the rows and columns to determine the non-zeroes of the matrix. Our heuristic approach is a greedy approach.

In Chapter 5, we present an iterative approach to group the rows and the columns. Here, we first randomly assign color to rows and columns, then we try to determine whether the coloring is appropriate. If not, then we reassign colors.

In Chapter 6, we provide experimental results that demonstrate the efficacy of our algorithms. We compare our lower bound results with the lower bound results of [11] and [21]. We also compare our coloring heuristic results with the works of [11],[14],[21]. Results of our iterative algorithm is also discussed here.

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

Chapter 2

Preliminaries

Sparse Matrix computation problems can be modelled using graphs. From graphs, we can reveal valuable information which can be used to determine different properties of the matrices. In this chapter, we introduce basic graph terminologies.We also discuss the compressed data structure that we have used instead of $m \times n$ sized adjacency matrix data structure.

2.1 Graph

A *graph* G is an ordered pair (V,E) where V is a finite and non-empty set of vertices/nodes and E is a set of edges.

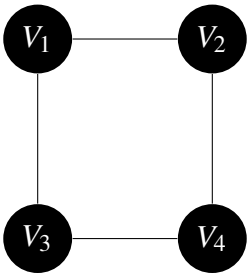

Figure 2.1: Drawing of a graph $G = (V,E)$, where the vertex set $V = \{v_1, v_2, v_4, v_3\}$ and edge set $E = \{\{v_1, v_2\}, \{v_2, v_4\}, \{v_4, v_3\}, \{v_3, v_1\}\}$. The vertices are shown in circles and edges are represented as lines connecting two vertices.

Two vertices that are connected by an edge are called *adjacent nodes*. The *degree* of a vertex v, denoted by $deg(v)$ is the total number of vertices adjacent to v. A *path* of length l, denoted by *l-path* is a sequence $\{v_1, v_2, \ldots, v_{l+1}\}$ of distinct vertices in G such that v_i and v_{i+1} are adjacent for $1 \leq i \leq l$. In this thesis, we consider simple graphs i.e. graphs without multiple edges and self-loops.

2.2 Clique of a graph

A complete graph is a simple undirected graph where every vertex is connected with every other vertex. A *Sub-graph* $G'(V',E')$ of a graph $G(V,E)$ is a graph where $V' \subset V$ and $E' \subset E$. A clique of graph G is a complete sub-graph of G. A maximum clique is defined as the clique with the maximum number of vertices. The size of a clique is measured in the number of vertices from the clique.

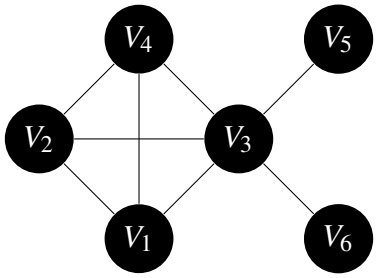

Figure 2.2: A graph containing a clique of size 4

In Figure 2.2 , vertices V_1, V_2, V_3, V_4 are connected with each other to form a clique. This is the maximum clique of the graph.

2.3 Bipartite graph

A *Bipartite graph* $G_b = (U \cup V, E)$ contains two disjoint set of vertices U and V, where every edge has one endpoint in U and the other endpoint in V.

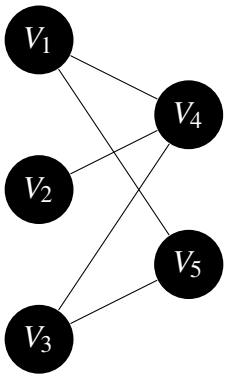

Figure 2.3: A bipartite graph $G_b = (U \cup V, E)$, where $U = \{v_1, v_2, v_3\}$ and $V = \{v_4, v_5\}$ and the edge set $\{\{v1, v4\}, \{v1, v5\}, \{v2, v4\}, \{v3, v4\}, \{v3, v5\}\}$.

2.4 Tree

A *tree* is a data structure made up of nodes where each node is a data structure consisting of a value and a list of references to other nodes(the ”children”). A *Binary tree* is a tree where each node has at-most two children.

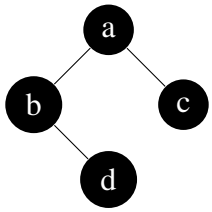

Figure 2.4: A binary tree where every node has 0 child(node d,c), 1 child(node b) or 2 children (node a)

2.5 Binary heap tree

A binary heap tree is a data structure with a binary tree topology, satisfying the following conditions:

- The tree is a complete binary tree, i.e. each level except (may be) the last level are full with vertices, if the last level is not full then it fills out vertices from left side.
- The value stored in any parent node of the tree is greater/equal to the value stored at the children (max-heap) or less than/equal to the value of the children(min-heap).

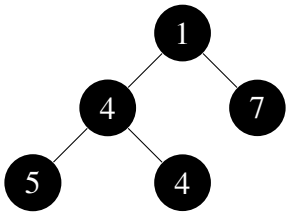

Figure 2.5: A min-heap tree

2.6 Graph coloring

Graph coloring is an assignment of colors to the vertices such that no two adjacent vertices are assigned the same color. A p-coloring of a graph $G = (V, E)$ is a function

$\phi : V \rightarrow \{1,2,\ldots,p\}$ 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.

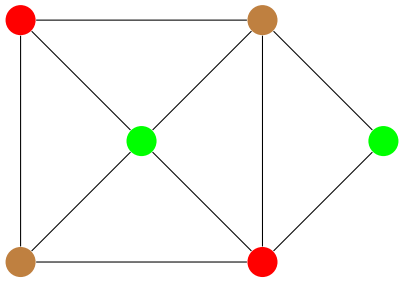

Figure 2.6: 3-coloring of a graph

Definition 1. Let $G_b = (U \cup V, E)$ be a bipartite graph partitioned into two disjoint set of vertices U and V. A coloring ϕ is a *path p-coloring* of G_b, if every 3-length path in G_b uses atleast 3 colors and

$$\{\phi(u) : u \in U\} \cap \{\phi(v) : v \in V\} = \emptyset \tag{2.1}$$

Definition 2. A partition of the rows and columns of the matrix A is known as *row-column consistent partition*(or *bi-directional partitioning*), if we can divide the non-zeroes of the matrix into groups of either rows or either columns in such away that for every non-zero a_{ij} of A, either column c_j is in a group where no other column in its group has a non-zero in row r_i or row r_i is in a group where no other row in its group has a non-zero in column c_j.

A special case of Definition 2 is to restrict the partitioning, to the columns of the matrix A or A^T. The resulting partitioning is also known as *unidirectional partitioning*.

2.7 Representation of partitioning problem as a graph coloring problem

A Sparse matrix $A \in \Re^{m \times n}$ can be represented as a bipartite graph $G_b(A)$. The vertex sets of the bipartite graph correspond to the rows and columns of the matrix such that $U = \{c_1, c_2, \ldots, c_n\}$, where c_j is the j-th column and $V = \{ r_1, r_2, \ldots, r_m\}$, where r_i is the

i-th row. For every non-zero a_{ij} in matrix A, there is an edge $\{r_i, c_j\} \in E$. Let us consider the following sparse matrix.

$$A = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \tag{2.2}$$

The corresponding bipartite graph for matrix A will be

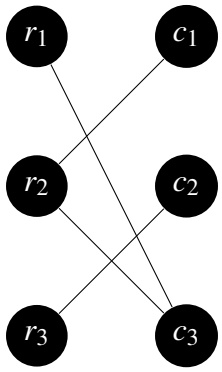

Figure 2.7: A bipartite graph $G_b = (U \cup V, E)$ representing the sparse matrix A in Equation 2.2 where vertex set $U = \{r_1, r_2, r_3\}$ represents rows and $V = \{a_1, a_2, a_3\}$ represents columns of A. Non-zeroes $\{a_{13}, a_{21}, a_{23}, a_{32}\}$ are represented by edge set $\{\{r_1, c_3\}, \{r_2, c_1\}, \{r_2, a_3\}, \{r_3, a_2\}\}$.

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 the number of edges $|E| = nnz(A)$.

2.7.1. Theorem. *Let A be an $m \times n$ matrix. A mapping Φ induces a row-column consistent partition of matrix A if and only if ϕ is a path p-coloring of $G_b(A)$.*

Proof of this theorem was given in [16].

Graph formulation of partitioning problems offer several advantages. First, it is more convenient to analyze the computational complexity when a partitioning problem is modelled using a graph. Furthermore, graph algorithms that are known to produce "good coloring" can be applied to the partitioning problems. Unfortunately, both the standard p-coloring and the path p-coloring are NP-Complete. For a comprehensive introduction to

computational intractability we refer to [9]. In our thesis, one of our major goal is to obtain a row-column consistent partition where the sum of row groups and column groups is minimized.

2.8 Graph coloring methods

We have used two algorithms, one based on heuristic approach and the other based on iterative approach to color the vertices of the graph. Our heuristic approach yields a result in polynomial time. In the Iterative approach, we have introduced a randomization technique to color the vertices of the graph.

2.8.1 Heuristic method

Heuristic methods are those methods which are designed to solve problems more quickly when classical methods are slow. It is not guaranteed that, we will get an optimal solution of the problem using heuristic methods.

The heuristic approach that we have used in this thesis is based on a greedy constructive algorithm. We partition the vertices into $V_1, V_2, \ldots, V_p$ independent sets and construct p color groups. The first element of every set is the row or column which has the maximum number of non-determined non-zeroes and is not grouped yet. While creating a new set, we make a temporary group. Nodes in the temporary group will be the nodes which are connected with the elements of the currently computing set by a 2-length path. Every time we insert a node into the set, we update the temporary group. The vertex which has the maximum 2-length path connection with the vertices in temporary group will be the next element of the set. When there is no such element to insert into the set, then we start a new set using the same procedure.

2.8.2 Iterative method

An iterative method is a problem solving method which generates a sequence of improving approximate solutions. We have introduced a randomization technique to color the

nodes in order to partition the non-zeroes of the matrix in our iterative algorithm. At first we randomly color all the nodes of the matrix. Then calculate all the 3-length paths of the matrix and determine the number of colors in each 3-length path. After that, we build a min heap tree based on the numbers of colors in each path. If the number of colors of the path that reside in the root node is not greater than or equal to 3, then randomly re-color a row or column from the path at root node and delete all the paths from the heap tree which contains that row or column, we re-insert those nodes into the heap tree again. We continue our iteration until the root of the heap tree becomes three or more.

2.9 Data structure

A sparse matrix has many zeroes which remain unused. So instead of using $m \times n$ matrix, a different data structure was proposed in [14] which exploits the sparsity of the matrix. The data structure that is used in this thesis takes only $3 \times nnz(A) + m + n + 2$ memory locations. The total data structure is divided into two parts: Compressed Column Storage(CCS) and Compressed Row Storage(CRS).

2.9.1 Compressed column storage

In compressed column storage, row indices of each column are stored into an array named as *row_ind*. Another array *col_ptr* contains the starting index of each column. Non-zero values are stored into another array named *data*. Row indices of each column j are in between $row\_ind[col\_ptr[j]]$ and $row\_ind[col\_ptr[j+1]-1]$. Hence total amount of memory required to store compressed column storage is $2nnz(A)+m+1$.

2.9.2 Compressed row storage

Like Compressed Column Storage, Compressed Row Storage uses *col_ind* to store column indices of each row and *row_ptr* to represent starting index of each row. Column indices of each row i can be found in between $col\_ind[row\_ptr[i]]$ and $col\_ind\ [\ row\_ptr[i+1]-1]$. Compressed Row storage uses $nnz(A)+n+1$ memory locations.

Let us the consider the following matrix:

$$A = \begin{bmatrix} a_{11} & 0 & 0 & a_{14} & 0 \\ 0 & a_{22} & 0 & 0 & a_{25} \\ 0 & a_{32} & 0 & a_{34} & 0 \\ a_{41} & 0 & 0 & 0 & a_{45} \\ 0 & 0 & a_{53} & 0 & 0 \end{bmatrix} \tag{2.3}$$

The corresponding data structure of the matrix is as follows:

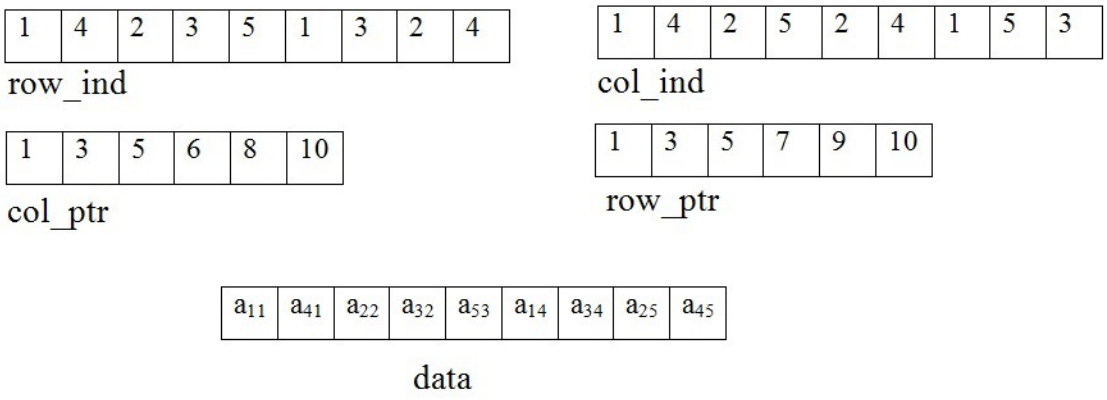

Figure 2.8: Compressed data structure for the matrix in Equation 2.3

Hence a total of $3nnz(A)+m+n+2$ memory locations used instead of $m \times n$ memory.

2.10 Conclusion

In this chapter, we have introduced basic graph terminologies that we will use in the later chapters. We also discussed the compress data structure to store sparse matrix efficiently.

Chapter 3

Lower Bound

In this chapter, we derive a new lower bound on the number of groups required to determine the non-zeroes of a Jacobian matrix.

3.1 Background

If we can calculate a good lower bound, then we can measure how much improvement an algorithm requires to achieve the optimal result. If there is a gap between the lower bound and the best algorithm, then it is required either to improve the lower bound of the problem or to increase the efficiency of the algorithm. In the case of a matrix partitioning problem, we want to find the least number of groups that one must have in a partition to determine the non-zeroes of a matrix.

DSM [4] is a software program that is used to determine Jacobian matrices. **DSM** interprets the matrix as a graph, where nodes are the columns of the matrix and there will be an edge between two columns if the columns have non-zero in the same row. The size of a clique is a lower bound on the chromatic number of a graph. A simple but good lower bound on the size of the largest clique is given by the maximum number of non-zeroes in any row of the matrix.

In a row-column consistent partition, in a column group no two columns can share non-zero in the same row. Therefore, the maximum number of non-zeroes can be determined by a column group can not be more than the number of rows m. Similarly, the total number of non-zeroes that can be determined by a row group is less than or equal to the number of columns n. So we can say that, the maximum number of non-zeroes that can be determined by a group is equal to $max\{m,n\}$. If there are p groups where each group can determine

at-most $max\{m,n\}$ non-zeroes, then we must have $p \geq \frac{nnz}{max\{m,n\}}$. David Juedes and Jeffrey Jones in their paper [21] view the matrix as a bipartite graph and calculate lower bound by taking the ratio of $|E|$ and $max\{|V_1|,|V_2|\}$, where $|E|$ is the number of edges in the graph and $|V_1|$ and $|V_2|$ are the two sets of vertices. This idea was independently proposed in [15].

In bi-directional partitioning a lower bound needs to incorporate both dense rows and dense columns. The lower bound that we have proposed in this thesis is based on the observation that dense rows and dense columns induce dense sub-matrices. To identify the dense sub-matrices, we permute the columns and rows of the given Jacobian matrix such that dense rows and columns are moved towards the top left corner of the matrix.

3.2 Re-arrangement based on degree sorting

The number of non-zeroes in a row or column is defined as the degree of that row or column. In the degree sorting algorithm, we permute the rows and the columns in the non-increasing order of the number of the non-zeroes. For that purpose, we first permute the rows based on the degree of the rows. After that, we again permute the resulting matrix depending on the number of non-zeroes in each column. The algorithm for degree sorting is given below:

DEGREE-SORTING()
1 **for** $i = 1$ **to** m
2 $\quad row\_degree[i] = row\_ptr[i] - row\_ptr[i-1]$
3 **for** $i = 1$ **to** n
4 $\quad column\_degree[i] = col\_ptr[i] - col\_ptr[i-1]$
5 Permute the rows of the matrix in non-increasing order based on the value of *row_degree* and rearrange the matrix based on the permutation order of the rows.
6 Permute the columns in non-increasing order based on the value of *column_degree* of the resultant matrix and rearrange the matrix based on the permutation order of the columns.

Figure 3.1: Algorithm for the permutation of rows and columns by using degree sorting

In the algorithm DEGREE-SORTING(), first we compute and store the nuber of non-

zeroes of each row in *row_degree*. In *column_degree*, we store the number of non-zeros in each column. Then we sort the rows in non-increasing order based on the value of *row_degree* and rearrange the matrix based on the permutation order of the rows. After that, we sort the columns in non-increasing order based on the value of *column_degree* and rearrange the matrix based on the permutation order of the columns.

Let us discuss the algorithm with an example. Consider the following matrix:

$$A = \begin{bmatrix} \times & 0 & \times & 0 & 0 & \times \\ 0 & 0 & 0 & \times & 0 & \times \\ \times & \times & \times & 0 & \times & 0 \\ \times & 0 & \times & 0 & 0 & 0 \\ 0 & \times & 0 & 0 & \times & \times \\ \times & 0 & \times & 0 & 0 & 0 \end{bmatrix} \tag{3.1}$$

Here $\times$ represents the non-zero elements of matrix A. Total number of non-zeroes in the rows of A are 3, 2, 4, 2, 3 and 2. Hence the values of *row_degree* for the rows of the matrix are:

3	2	4	2	3	2

On the other side, the number of non-zeroes in each column of matrix A are 4, 2, 4, 1, 2 and 3. So *column_degree* will be:

4	2	4	1	2	3

r_3 has the highest *row_degree* value, which is 4, then in sequence r_1, r_5, r_2, r_4 and r_6. If we sort the value of *row_degree* in non-increasing order then we get:

4	3	3	2	2	2

And the rearrangement of the rows of matrix A based on *row_degree* value will be

r_3	r_1	r_5	r_2	r_4	r_6

If we permute the rows based on the new arrangement of rows, matrix A will then become,

$$A = \begin{bmatrix} \times & \times & \times & 0 & \times & 0 \\ \times & 0 & \times & 0 & 0 & \times \\ 0 & \times & 0 & 0 & \times & \times \\ 0 & 0 & 0 & \times & 0 & \times \\ \times & 0 & \times & 0 & 0 & 0 \\ \times & 0 & \times & 0 & 0 & 0 \end{bmatrix} \tag{3.2}$$

Permuting the rows based on *row_degree* does not change the value of *column_degree*. For the resultant matrix in the Equation 3.2, we have *column_degree* like

4	2	4	1	2	3

For the resultant matrix, c_1 and c_3 have the highest number of non-zeroes, then in sequence c_6, c_2,c_5 and c_4. The sorted order of the *column_degree* will be:

4	4	3	2	2	1

And the arrangement of columns based on the sorted order of *column_degree* is

c_1	c_3	c_6	c_2	c_5	c_4

Hence, based on the new column arrangement our final matrix will be:

$$A = \begin{bmatrix} \times & \times & 0 & \times & \times & 0 \\ \times & \times & \times & 0 & 0 & 0 \\ 0 & 0 & \times & \times & \times & 0 \\ 0 & 0 & \times & 0 & 0 & \times \\ \times & \times & 0 & 0 & 0 & 0 \\ \times & \times & 0 & 0 & 0 & 0 \end{bmatrix} \tag{3.3}$$

.

3.3 Re-arrangement based on lexicographical sorting

Lexicographical sorting of the rows means rearrangement of the rows based on the column position of the non-zeroes in each row. Similarly lexicographical sorting of the columns represents the rearrangement of the columns based on the row position of the non-zeroes in each column.

3.3.1 Lexicographical ordering of rows

Rows are permuted based on the column position of the non-zeroes in each row. For the matrix in Equation 3.1, column position of the non-zeroes of the rows are $r_1 = \{1,3,6\}$, $r_2 = \{4,6\}$, $r_3 = \{1,2,3,5\}$, $r_4 = \{1,3\}$, $r_5 = \{2,5,6\}$ and $r_6 = \{1,3\}$. If we lexicographically sort the rows, then the row arrangement becomes $r_3 = \{1,2,3,5\}$, $r_4 = \{1,3\}$, $r_6 = \{1,3\}$, $r_1 = \{1,3,6\}$, $r_5 = \{2,5,6\}$ and $r_2 = \{4,6\}$. The algorithm to sort the rows by using lexicographical ordering is given in Figure 3.5.

The first step in order to rearrange the rows based on the column position of the non-zereos is to find the row which has the maximum number of non-zeroes among the rows. We need to continue our iteration upto the maximum number of non-zero times in order to compare the non-zero column indices of every row. The procedure to find the maximum

number of non-zeroes in a row of the matrix is given in Figure 3.2.

```
MAXIMUM-NUMBER-OF-NONZERO-IN-A-ROW ()
1  for i = 1 to m
2       row_degree[i] = row_ptr[i]− row_ptr[i−1]
3  max_value = 0
4  for i = 1 to m
5       if row-degree[i] >max_value
6            max_value = row-degree
```

Figure 3.2: Algorithm to find maximum number of non-zero in a row of the matrix

In procedure MAXIMUM-NUMBER-OF-NONZERO-IN-A-ROW(), we calculate the number of non-zeroes in each row and store it in *row-degree*. Variable *max_value* stores maximum number of non-zeroes in a row. Initially, *max_value* is assigned to 0. If *max_value* is less than *row-degree* value of the corresponding row, then *max_value* gets *row-degree* value of that row.

Lexicographically-Sorted-Row keeps a record of the permuting sequence of the rows. At beginning, the rows are in normal sequence i.e.

r_1	r_2	r_3	r_4	r_5	r_6

While comparing column position of the rows, we first check whether the last non-zero column position of the rows is the same or not. Array *Rows-Comparable-or- not* determines whether previous non-zero position of the two rows is same or not. If values in the consecutive index of the *Rows-Comparable-or-not* are same then we compare those two rows, otherwise not. For the first iteration, there is no previous index. So we compare first non-zero column position of every row with one another. As there is no previous index for first column position, so we put 1 in every index of the *Rows-Comparable-or-not*.

1	1	1	1	1	1

Row r_3 has the maximum number of non-zeroes and it is 4. So we have to do our iteration 4 times in order to sort the rows lexicographically. Non-zero column indices of

each row are stored in *col_ind* and *row_ptr* tells the starting index of each row in *col_ind*. After each iteration, rows interchange their position in *Lexicographically-Sorted-Row*. So we need to keep track of each row and the corresponding non-zero column position of the row for that iteration. The procedure in Figure 3.3 finds non-zero column index of the corresponding row for an iteration. It takes two arguments. The first argument *index_value* is the index of *Lexicographically-Sorted-Row* and the second argument *iteration* is the number of iteration, i.e., which non-zero column index of the row that we are looking for. Variable *row-number* stores row number, and *current-row-degree* stores the number of non-zeroes in the corresponding row. Line 3 of Figure 3.3 finds the starting index of the row that resides in *row_ptr*. Then it compares *iteration* value with the degree of that row which is stored in *current-row-degree*. If the value of *iteration* is less than or equal to *current-row-degree* then it returns the corresponding column indices from *col_ind*, otherwise it returns -1.

```
COLUMN-INDICES-CALCULATION(index_value,iteration)
1  row-number = Lexicographically-Sorted-Row[index_value ]
2  current-row-degree = row-degree[ row-number −1]
3  current-row-starts = row_ptr [ row-number −1]
4  current-column-index-position = current-row-starts + iteration
5  if iteration ≤ current-column-index-position
6        current-column-indices = col_ind [ current-column-index-position ]
7  else current-column-indices = −1
```

Figure 3.3: Algorithm for the column position calculation of a row

col_ind and *row_ptr* for the matrix in Equation 3.1 are as follows

col_ind	1	3	6	4	6	1	2	3	5	1	3	2	5	6	1	3

row_ptr	1	4	6	10	12	15	17

In order to find the 2*nd* non-zero column index of the row r_4 of matrix A in Equation 3.1, we first check the degree of r_4, which is 2. The starting index of r_4 can be found from *row_ptr*, which is 10. So index position of the second non-zero column index of r_4 will be in the $(10+2)-1 = 11th$ index of *col_ind* and it is 3.

In algorithm LEXICOGRAPHICAL-SORT-FOR-ROWS(), **for** loop in line 5 runs for the maximum number of non-zeroes in a row of the matrix to ensure the comparison of each non-zero column indices in a row. **for** loop in lines 6 and 7 are used to compare non-zero column indices between consecutive rows. Variable *current-row-weight* and *previous-row-weight* check *Rows-Comparable-Or-Not* value between consecutive rows. If both the values are equal, then we look for the non-zero column index of the corresponding rows by using the COLUMN- INDICES-CALCULATION() algorithm in Figure 3.3. If first non-zero column index of the row in current index of *Lexicographically- Sorted-Row* is less than the first column index of the row in the previous index of *Lexicographically-Sorted-Row*, then interchange the two row positions in *Lexicographically- Sorted-Row*.

For the first iteration, first non-zero column indices of the rows in matrix *A* of Equation 3.1 are 1, 4, 1, 1, 2 and 1. If we sort the rows of matrix *A* based on the increasing order of the first non-zero column indices, then *Lexicographically-Sorted-Row* becomes

r_1	r_3	r_4	r_6	r_5	r_2

After the first iteration, first non-zero column indices of the rows in *Lexicographically-Sorted-Row* are in sorted order. The next step is to update *Rows-Comparable-or-not*, so that we can start our next iteration. The algorithm in Figure 3.4 updates the *Rows-Comparable-or-not* values of each row. First we assign 1 to the first index of *Rows-Comparable-or-not*. Then we start an iteration to calculate the *Rows-Comparable-or-not* value for the rest of the rows. In each iteration, we calculate the corresponding non-zero column index of

current row and previous row using the algorithm in Figure 3.3. If the non-zero column index of consecutive rows stored in *Lexicographically-Sorted-Row* are equal and not -1 then we assign *Rows-Comparable-or-not* value of the previous index into current index, otherwise assign current index number of *Rows-Comparable-or-not* into the index of *Rows-Comparable-or-not*.

```
UPDATE-ROWS-COMPARABLE-OR-NOT(iteration)
1  Rows-Comparable-or-not[1] = 1
2  for k = 2 to row
3      current-column-indices = COLUMN-INDICES-CALCULATION(k, iteration )
4      previous-column-indices = COLUMN-INDICES-CALCULATION(k − 1, iteration )
5      if current-column-indices ≠ −1&& previous-column-indices ≠ −1
6          if current-column-indices == previous-column-indices
7              Rows-Comparable-or-not[k] = Rows-Comparable-or-not[k − 1]
8          else Rows-Comparable-or-not[k] = k
9      else Rows-Comparable-or-not[k] = k
```

Figure 3.4: Algorithm for up-gradation of variable in the array Rows-Comparable-or-not

First non-zero column index of the first four rows (r_1, r_3, r_4 and r_6) of *Lexicographically-Sorted- Row* are same and it is 1. r_2 is in the fifth index of *Lexicographically-Sorted-Row* and its first non-zero column index is 2. Row in the last index of *Lexicographically-Sorted-Row* is r_5, which has first non-zero column index at 4. So *Rows-Comparable-or-not* value for the first four indices will be the same and it is 1. The first non-zero index of the row in the fifth index of *Lexicographically-Sorted-Row* is not equal to the first non-zero column index of the row in the fourth index of *Lexicographically-Sorted-Row*. Therefore, we assign the value of index position, which is 5 into the fifth index of *Rows-Comparable-or-not*. Similarly, in the same way, the last index of *Rows-Comparable-or-not* gets the value 6. So updated *Rows-Comparable-or-not* will be

1	1	1	1	5	6

Values in the first four indices of *Rows-Comparable-or-not* are the same, so for the next iteration, we compare the second non-zero column index between the first four rows

of *Lexicographically-Sorted-Row*. Values in the fifth and sixth index of *Lexicographically-Sorted-Row* are unique. So rows in the last two indices retain their position for the next iteration. For the second iteration, we will now compare second non-zero column index of each row. If any of the rows has degree less than 2, then we assume -1 for the non-zero column index of that row. After the second iteration *Lexicographically-Sorted-Row* value will be

r_3	r_4	r_6	r_1	r_5	r_2

and the *Rows-Comparable-or-not* value will be

1	2	2	2	5	6

In the same way, for the next iteration third non-zero column index of r_3 is 3. r_4 and r_6 have only two non-zero indices, so their value will be -1. Third non-zero column index of r_1 and r_5 is 6. As r_2 has only two non-zero column indices, so its value will be -1. *Rows-Comparable-or-not* value for r_4, r_6 and r_1 are the same and rest are different. So r_3, r_5 and r_2 will keep their previous position. As r_4 and r_6 have only two non-zero column indices, we do not need to re-permute any rows for this iteration. Therefore *Lexicographically-Sorted-Row* value will be

r_3	r_4	r_6	r_1	r_5	r_2

and *Rows-Comparable-or-not* value will be

1	2	3	4	5	6

For the last iteration *Rows-Comparable-or-not* value of each row is different, so we do not need to re-permute the rows. Finally *Lexicographically-Sorted-Row* value will be

r_3	r_4	r_6	r_1	r_5	r_2

```
LEXICOGRAPHICAL-SORT-FOR-ROWS()
 1  maximum-degree = MAXIMUM-NUMBER-OF-NONZERO-IN-A-ROW ()
 2  for i = 1 to row-number
 3      Lexicographically-Sorted-Row[i] = i
 4      Rows-Comparable-Or-Not[i] = 1
 5  for row-index = 1 to maximum-degree
 6      for i = 2 to row-number
 7          for j = i downto 1
 8          current-row-weight = Rows-Comparable-Or-Not [j]
 9          previous-row-weight = Rows-Comparable-Or-Not [j − 1]
10          if current-row-weight == previous-row-weight
11              current-column-indices = COLUMN-INDICES-CALCULATION(j,
                                                        row-index )
12              previous-column-indices = COLUMN-INDICES-CALCULATION(j − 1)
13              if current-column-indices ≠ −1&& previous-column-indices ≠ −1
14                  if current-column-indices < previous-column-indices
15                      SWAP-ROWS( Lexicographically-Sorted-Row [j],
                                        Lexicographically-Sorted-Row [j − 1])
16              if current-column-indices == previous-column-indices
17                  if current-row-degree < previous-row-degree
18                      SWAP-ROWS( Lexicographically-Sorted-Row [j],
                                        Lexicographically-Sorted-Row [j − 1])
19      UPDATE-ROWS-COMPARABLE-OR-NOT( row-index )
```

Figure 3.5: Algorithm to sort the rows of a matrix lexicographically

Hence after lexicographical ordering of the rows, matrix A of Equation 3.1 will become

$$A = \begin{bmatrix} \times & \times & \times & 0 & \times & 0 \\ \times & 0 & \times & 0 & 0 & 0 \\ \times & 0 & \times & 0 & 0 & 0 \\ \times & 0 & \times & 0 & 0 & \times \\ 0 & \times & 0 & 0 & \times & \times \\ 0 & 0 & 0 & \times & 0 & \times \end{bmatrix} \tag{3.4}$$

3.3.2 Update data structure

After sorting the matrix based on lexicographical ordering of the non-zero column position in rows, row indices in each column change position from their original position in Equation 3.1. So we need to rearrange the row compressed and column compressed structure before we start lexicographical ordering of the columns. For that purpose, we create a temporary column compressed and row compressed data structure that stores the value of *col_ind* and *row_ptr* based on the permutation order of the rows in *Lexicographically-Sorted-Row*. After that, we assign the values in temporary data structure into the original row compressed structure. From the compressed row structure, we then calculate the value of *row_ind* and *col_ptr*.

Updated *row_ptr* and *col_ind* of the matrix in Equation 3.4 are

col_ind	1	2	3	5	1	3	1	3	1	3	6	2	5	6	4	6

row_ptr	1	5	7	9	12	15	17

And new *col_ptr* and *row_ind* will be

row_ind	1	2	3	4	1	5	1	2	3	4	6	1	5	4	5	6

col_ptr	1	5	7	11	12	14	17

3.3.3 Lexicographical ordering of columns

The next step in lexicographical sorting is to sort the matrix based on the non-zero row position of the columns. The procedure for lexicographical ordering of columns is

almost the same as the ordering of the rows. The main difference is we have to consider the non-zero row indices in columns instead of the non-zero column indices in rows. The row position in each column of the new matrix in Equation 3.4 are $c_1 = \{1,2,3,4\}$, $c_2 = \{1,5\}$, $c_3 = \{1,2,3,4\}$, $c_4 = \{6\}$, $c_5 = \{1,5\}$ and $c_6 = \{4,5,6\}$. After sorting the columns based on row position in each column, the column ordering becomes $c_1 = \{1,2,3,4\}$, $c_3 = \{1,2,3,4\}$, $c_2 = \{1,5\}$, $c_5 = \{1,5\}$, $c_6 = \{4,5,6\}$ and $c_4 = \{6\}$. So the final matrix configuration after lexicographical sorting will be:

$$A = \begin{bmatrix} \times & \times & \times & \times & 0 & 0 \\ \times & \times & 0 & 0 & 0 & 0 \\ \times & \times & 0 & 0 & 0 & 0 \\ \times & \times & 0 & 0 & \times & 0 \\ 0 & 0 & \times & \times & \times & 0 \\ 0 & 0 & 0 & 0 & \times & \times \end{bmatrix} \tag{3.5}$$

3.4 Lower bound calculation

The two types of sorting (degree sorting and lexicographical sorting) that we have shown in previous sections bring dense sub-matrices to the upper left side of the matrix. In this section, we describe a procedure that calculates the lower bound for the resultant matrix. At first we calculate the ratio of total number of non-zeroes in the sub-matrix and the number of rows in the sub-matrix. Maximum of all such ratios is a lower bound on the number of groups in a bi-directional determination of a Jacobian matrix.

Algorithm LOWER-BOUND() starts by computing the minimum of m and n and stores the value in variable *min*. We need to check the ratio $\frac{nnz_i}{i}$ for every $i \times i$ sub-matrix, where $i = 1,2,\ldots,min\{m,n\}$ and nnz_i denotes the number of non-zeroes in the $i \times i$ sub-matrix. For each sub-matrix, variable *count* stores the number of non-zeroes in the sub-matrix.

Variable *lower-bound* stores the maximum of $\frac{nnz_i}{i}$.

```
LOWER-BOUND()
1  min = (row<column)? row : column
2  lower-bound = 0
3  for i = 1 to min
4       count = total number of non-zero in i × i matrix
5       ratio =ceil( count /i)
6       if ratio >lower-bound
7            lower-bound = ratio
```

Figure 3.6: Algorithm to calculate lower bound of a matrix

In degree-sorting the number of non-zeroes in 1×1, 2×2, 3×3, 4×4, 5×5, 6×6 sub-matrix of matrix A in Equation 3.3 are 1, 4, 6, 9, 13, 16. Lower bound for the sub-matrices are 1, 2, 2, 3, 3,and 3. Among them the maximum is 3. So lower bound of A by using degree sorting is 3. In the same way lower bound of matrix A in equation 3.5 by using lexicographical sorting is also 3.

3.5 Conclusion

In this chapter, we have provided an estimation of the lower bound to calculate the minimum number of groups needed to determine the non-zeroes of a Jacobian matrix. The two types of sorting, degree sorting and lexicographical sorting, bring dense sub-matrices to the upper left corner of the matrix.

Chapter 4

Heuristic Approach

In this chapter, we describe our heuristic algorithm that groups columns and rows to determine the non-zeroes of the sparse Jacobian matrix. In a row group, if two rows have non-zeroes in the same column then the non-zeroes are not determined by that row group. These non-zeroes must be determined in a column group. Similarly, if more than one column from the same group has non-zeroes in the same row, then the non-zeroes cannot be determined by that column group. A row group determines these non-zeroes. To verify that a given bi-directional partition determines a sparse matrix, we state a proposition linking the row-column indices of the non-zero entries determined by the grouping algorithms. A constructive proof is provided which has been programmed to verify the bipartition constructed by our grouping algorithms. We will discuss step by step procedure of our algorithm in the following sections, and finally combine all the procedures to build our final heuristic algorithm.

4.1 Background

No exact algorithm is known to partition the sparse matrices to determine the non-zeroes using the minimum number of groups. Coleman and More [4] first proposed that the partitioning problem can be solved by using graph coloring algorithms and they developed a graph from the matrix where the nodes are the columns of the matrix, and there will be an edge between two nodes if two columns have atleast one non-zero in same row. Then they sorted the vertices of the graph using different ordering techniques such as Largest First Ordering(LFO), Smallest Last Ordering (SLO) and Incidence Degree Ordering (IDO), and used a sequential algorithm on the ordered vertices to color the graph. Later Hasan

introduced another ordering technique inspired by Recursive Largest First (RLF) ordering in [14]. The preceding authors used only columns to make groups to partition the matrix. It has been seen that when there are dense rows or dense columns in the matrix then coloring used by the aforementioned procedures need more groups to determine the non-zeroes. On the other hand, a Bi-directional approach handles the situation better . Mini Goyal in [11] developed bi-directional ordering of the algorithms used in [4] and found the results were better than unidirectional partitioning. Judes and Jones in [21] developed an approximation algorithm that calculates distance-2 independent sets from both row and column sides, and colored the vertices from the same set with the same color. In our thesis, we develop a bi-directional partitioning algorithm. Our algorithm is inspired by the recursive partitioning and graph coloring problem. In our approach, the first node of every group is the row or column that has the maximum number of non-zeroes. Then we create a temporary group and insert the vertices into the group that is connected to the nodes in the calculating group with a path length of 2. The remaining nodes of this group are selected as the nodes that have the largest connection with the temporary group by a path length of 2.

4.2 Maximum degree calculation

The first step to determine the non-zeroes of a matrix is to find which row or column contains the maximum number of non-zeroes. The degree of a row is defined as the number of non-zeroes in a row. Column degree is defined as the number of non-zeroes in the columns.

The algorithm DEGREE-CALCULATION() in Figure 4.1 calculates degree of rows and columns. Lines 1 - 2 calculate degree of each row and *column_degree* of each column is calculated by using lines 3- 4 of the algorithm.

Algorithm MAXIMUM-DEGREE-CALCULATION() in Figure 4.2 searches the row or column that has the maximum degree and finds the number of non-zeroes in that row or column. Variable *node* stores which node (either row node or column node) has maximum

```
DEGREE-CALCULATION()
1  for i = 1 to m
2      row_degree[i] = row_ptr[i]-row_ptr[i-1]
3  for i = 1 to n
4      column_degree[i] = col_ptr[i]-col_ptr[i-1]
```

Figure 4.1: Algorithm for degree calculation

number of non-zeroes and *direction* tells whether the node is row or column. The algorithm starts by initializing 0 to *max_degree*. It then checks *row_degree* of every row. If *row_degree* value of any row is greater than *max_degree*, then *max_degree* gets the value of *row_degree*, *node* gets the row number and "row" is assigned to *direction*. *max_degree* is then compared with *column_degree*. If any *column_degree* value of any column is greater than *max_degree* then *max_degree* gets the value of *column_degree* and the column number is assigned to *node*. Then we change the value of *direction* to "column".

```
MAXIMUM-DEGREE-CALCULATION()()
 1  max_degree = 0
 2  for i = 1 to m
 3      if max_degree < row_degree[i]
 4          max_degree = row_degree[i]
 5          node = i
 6          direction = "row"
 7  for i = 1 to n
 8      if max_degree < column_degree[i]
 9          max_degree = column_degree[i]
10          node = i
11          direction = "column"
```

Figure 4.2: Algorithm to select the node which has the maximum number of non-zeroes

Let us consider the sparse matrix in Equation 4.1:

$$A = \begin{bmatrix} \times & 0 & 0 & \times \\ 0 & \times & 0 & \times \\ 0 & 0 & \times & \times \\ \times & \times & \times & \times \end{bmatrix} \tag{4.1}$$

The *row_ptr*,*col_ind*,*col_ptr* and *row_ind* value of matrix A in Equation 4.1 are

col_ind	1	4	2	4	3	4	1	2	3	4

row_ptr	1	3	5	7	11

row_ind	1	4	2	4	3	4	1	2	3	4

col_ptr	1	3	5	7	11

Degrees of the rows of the matrix A in Equation 4.1 are 2,2,2,4 and degrees for each column are 2,2,2,4. Among the rows, r_4 has the highest number of non-zeroes. (i.e., r_4 has the highest degree.) So *max_degree* gets the value 4, *direction* is assigned as "row" and *node* gets the value 4 . Now *max_degree* is compared with *column_degree* of every column. There is no column whose value is greater than 4. So the value of *node* and *direction* will remain the same as before.

4.3 Distance-2 neighbor list calculation

Distance-2 neighbors of a vertex is defined as the vertices which are connected with the vertex by a path length of 2. In other words if $\{U,V,W\}$ is a path in a bipartite graph

G_b, where $\{U,V\}$ and $\{V,W\}$ are adjacent vertices of the graph, then U is the distance-2 neighbor of W and vice versa. In this section we will provide an algorithm that will calculate distance-2 neighbors for each node of the graph.

To calculate distance-2 neighbors of a row, we first check column indices of that row. Then we find row indices of the corresponding columns that are the column indices of the row. Distance-2 neighbor of the row will be row indices of the corresponding columns except the row itself. Similarly, to get distance-2 neighbor of a column, we first look at row indices of that column, then find column indices of the corresponding rows that are adjacent to the column. The columns indices of the neighbor rows of the column is the distance-2 neighbor of that column.

For the matrix in Equation 4.1, r_1 has column indices $\{1,4\}$. If we want to find the distance-2 neighbors of r_1, then we have to check row indices of the columns that r_1 has as column indices. Row indices for c_1 and c_4 are $\{1,4\}$ and $\{1,2,3,4\}$. So distance-2 neighbors for r_1 are $\{\{1,4\} \cup \{1,2,3,4\}\} \setminus \{1\}$ i.e. $\{2,3,4\}$. In the same way distance-2 neighbor list for r_2, r_3 and r_4 are $\{1,3,4\}$, $\{1,2,4\}$ and $\{1,2,3\}$.

```
ROW-DISTANCE-2-NEIGHBORS-CALCULATION()
1  for i = 1 to m
2      for j = row_ptr[i-1] to row_ptr[i]
3          column_no = col_ind[j]
4          if column_no ≠ 0
5              for k = col_ptr[column_no-1] to col_ptr[column_no]
6                  row_number = row_ind[k]
7                  if  (row_number ≠ 0&& (row_number ≠ i)
8                      row_2path_list[i] = row_number
```

Figure 4.3: Algorithm to find the distance-2 neighbors of rows in a matrix

In the algorithm ROW-DISTANCE-2-NEIGHBORS-CALCULATION(), *row_2path_list* stores the distance-2 neighbor of each row. The iteration in line 1 iterates to find the distance-2 neighbor list of every row. Then in line 2, we check column indices of the r_i . Variable

column_no stores column indices of the r_i one at a time. Then the iteration in line 5 finds row indices of the column, which is stored in *column_no*. Variable *row_number* stores row indices of the column . If *row_number* is not 0 or equal to i then the *row_number* will be distance-2 neighbor of r_i . *row_2path_list* stores distance2 neighbor list of each row.

The algorithm to calculate the distance-2 neighbor lists of the columns is given below:

```
COLUMN-DISTANCE-2-NEIGHBORS-CALCULATION()
1  for i = 1 to n
2      for j = col_ptr[i-1] to col_ptr[i]
3          row_no = row_ind[j]
4          if row_no ≠ 0
5              for k = row_ptr[row_no-1] to row_ptr[row_no]
6                  column_number = col_ind[k]
7                  if  (column_number ≠ 0&& (column_number ≠ i)
8                      column_2Path_list[i] = column_number
```

Figure 4.4: Algorithm to find the distance-2 neighbors of columns in a matrix

COLUMN-DISTANCE-2-NEIGHBORS-CALCULATION() in Figure 4.4 calculates distance-2 neighbor list for every column. *column_2Path_list* stores distance-2 neighbors list of every column . The first iteration in line 1 repeats for every column, then line 2 checks row indices for c_i. Variable *row_no* stores the row indices of c_i. Line 5 checks each column indices of the rows that are in *row_no*. The column indices are stored in *column_number* one at a time. If column_number is not 0 or not equal to c_i, then the column_number is stored in *column_2Path_list* for c_i.

4.4 Updating data structure

Whenever a row or column is assigned to a group, the next step is to identify all the non-zeroes of that row or column. For this purpose, we assign 0 to the position of *row_ind* and *col_ind* of the non-zeroes that are affected by that row or column. We reduce the degree of the rows and columns whose non-zeroes are marked as 0 by the amount of non-zero

position marked as 0. This is because every time we start a new group, we need to know how many non-zeroes in that row or column still needs to be determined.

Suppose r_4 is assigned to a group of the matrix A in Equation 4.1, then all the column indices of r_4 will be set to 0. After that we mark the position of the columns which have non-zero in r_4. For the first group c_1, c_2, c_3 and c_4 have non-zero in r_4. So the value of r_4 position in c_1, c_2, c_3, c_4 of *row_ind* is set to 0. As we mark all the non-zeroes of r_4 to 0, so the degree of r_4 becomes 0. Similarly, we marked one non-zero from c_1, c_2, c_3 and c_4, so the degree of c_1, c_2, c_3 and c_4 will now become 1,1,1,and 3 which was previously 2, 2, 2 and 4.

col_ind	1	4	2	4	3	4	0	0	0	0

row_ind	1	0	2	0	3	0	1	2	3	0

The algorithm DATAUPDATEBASEDONROW(*row_no*) in Figure 4.5 updates data structure when a row is assigned to a group. *row_no* is the row that is assigned to a group. The iteration in line 1 runs for column indices of the row which are stored in *row_no* and put the column in *column_no* one at a time. The iteration in line 3 checks row indices of the column stored in *column_no*. If the row indices is equal to *row_no* then set that *row_ind* to 0 and decrease *column_degree* of the column stored in *column_no* by 1. Then we check non-zeroes in *i*-th position of *col_ind*, whether it is marked as 0 or not. If it is not 0 then we increment the number of determined non-zeroes by 1 and turn the *col_ind* value for that non-zero to 0. Finally, we change the degree of the row stored in *row_no* to 0.

Algorithm DATAUPDATEBASEDONCOLUMN(*column_no*) in Figure 4.6 updates data structure when a column is assigned to a group. It is almost the same as updating data structure when a row is assigned to a group. Line 1 runs for row indices of the column that

```
DATAUPDATEBASEDONROW(row_no)
 1  for i = row_ptr[row_no−1] to (row_ptr[row_no]−1)
 2      column_no = col_ind[i]
 3      for j =col_ptr[column_no−1]to (col_ptr[column_no]−1)
 4          if row_ind[j] == row_no
 5              row_ind[j] = 0
 6              column_degree[column_no]= column_degree[column_no]−1
 7      if col_ind[i] ≠ 0
 8          nnz_count=nnz_count +1;
 9          col_ind[i] = 0
10  row_degree[row_no]= 0
```

Figure 4.5: Algorithm to modify the data structure when a row is assigned to a group

```
DATAUPDATEBASEDONCOLUMN(column_no)
 1  for i = col_ptr[column_no−1] to (col_ptr[column_no]−1)
 2      row_no = row_ind[i]
 3      for j =row_ptr[row_no−1]to (row_ptr[row_no]−1)
 4          if col_ind[j] == column_no
 5              col_ind[j] = 0
 6              row_degree[row_no]= row_degree[row_no]−1
 7      if row_ind[i] ≠ 0
 8          nnz_count=nnz_count +1;
 9          row_ind[i] = 0
10  column_degree[column_no]= 0
```

Figure 4.6: Algorithm to modify the data structure when a column is assigned to a group

is assigned to a group. For each row in *row_no*, line 3 checks column indices of that row. If there is any column indices which is equal to *column_no* then make that *col_ind* position as 0 and reduce the *row_degree* of that row by 1. Finally, it checks whether any row indices of the column in *column_no* is not yet assigned to 0. If any row indices is not 0, then change it to 0, increase the number of determined non-zeroes and make the degree of that column to 0.

4.5 uDegree calculation

While assigning row or column to a group, we have to make a temporary group that inserts row or column nodes which are distance-2 neighbors of the nodes in the group that we are currently calculating. uDegree of a node v is the number of nodes in the temporary group that has distance-2 neighbor connection with v. uDegree is mainly calculated for the nodes that are not yet assigned to a group and are not in the temporary group.

Let us consider the distance-2 neighbor list of the rows in a matrix as follows: r_1=$\{2,3\}$, r_2=$\{1,4,5\}$, r_3=$\{1,4\}$, r_4=$\{2,3,5\}$ and r_5=$\{2,4\}$. If we assign r_1 to a group, then r_2 and r_3 are distance-2 neighbors of r_1. So r_2 and r_3 will be in the temporary group. r_4 and r_5 are not assigned to a group and they are not in the temporary group. r_4 is connected with r_2 and r_3 in temporary group . So uDegree of r_4 is 2. Similarly r_5 is connected with only r_2 in the temporary group. So uDegree of r_5 is 1. The uDegree of the nodes that are assigned to a group or in the temporary group are marked as -1 for our calculation. So uDegree of r_1 , r_2 and r_3 are -1.

ROW-UDEGREECALCULATION(*row_no*)

```
1  row_uDegree[row_no] = −1
2  for i = 1 to row_2path_list[row_no].size
3       two_path_node = row_2path_list[row_no][i]
4       row_uDegree[ two_path_node] = −1
5  for i = 1 to row_2path_list[row_no].size
6       two_path_node = row_2path_list[row_no][i]
7       for j = 1 to row_2path_list[row_no].size
8            two_length_path_row = row_2path_list[two_path_node][j]
9            if row_uDegree[two_length_path_row] ≠ −1
10                row_uDegree[two_length_path_row]=
                          row_uDegree[two_length_path_row] +1
```

Figure 4.7: Algorithm to calculate the uDegree of rows

In the algorithm ROW-UDEGREECALCULATION(*row_no*), initially uDegree of every row is set to 0. Variable *row_no* stores the row that are currently assigned to a group. *row_uDegree* is used to store uDegree of every row. At first we assign *row_uDegree* of

```
COLUMN-UDEGREECALCULATION(column_no)
 1  column_uDegree[column_no] = −1
 2  for i = 1 to column_2path_list[column_no].size
 3       two_path_node = column_2path_list[column_no][i]
 4       column_uDegree[two_path_node] = −1
 5  for i = 1 to column_2path_list[column_no].size
 6       two_path_node = column_2path_list[column_no][i]
 7       for j = 1 to column_2path_list[two_path_node].size
 8            two_path_column = column_2path_list[two_path_node[[j]
 9            if column_uDegree[two_path_column[ ≠ −1
10                 column_uDegree[two_path_column] =
                              column_uDegree[two_path_column] +1
```

Figure 4.8: Algorithm to calculate the uDegree of columns

row_no to -1. Lines 2- 4 run for all the distance-2 neighbors of the row stored in *row_no* and set uDegree of each node to -1. All the nodes that we assigned -1 in the iteration used in lines 2- 4 are in the temporary group. Now lines 5- 10 iterate to get distance-2 neighbors of the rows that are in the temporary group. If uDegree of any row is not equal to -1 then we increase *row_uDegree* of corresponding row by 1.

The algorithm in Figure 4.8 calculates column uDegree of every column of the matrix. The concept used in the algorithm COLUMN-UDEGREECALCULATION(*column_no*) is almost the same as the algorithm used to calculate uDegree of the rows. Initially uDegree of every column is 0. Then we make uDegree of the column stored in *column_no* to -1. After that we change uDegree of all the distance-2 neighbors of *column_no* to -1. Finally we check distance-2 neighbor list of every distance-2 neighbor of *column_no*, if uDegree of any column is not set to -1, then increase its *column_uDegree* by 1.

4.6 Maximum uDegree calculation

In this section, we determine which row or column has the maximum uDegree. The node that has maximum uDegree greater than 0 will be assigned to the recent group after inserting the first row or column into that group. First row or column of a group is

calculated as the row or column that has the maximum degree and is not yet grouped.

The algorithm MAXIMUM-UDEGREE-IN-ROW() in Figure 4.9 compares uDegree of each row with *maximum_row_uDegree* which was initially assigned to 0. Whenever uDegree of any row is greater then *maximum_row_uDegree* then assign that row into *row_uNode* and update *maximum_row_uDegree* with uDegree of that row.

```
MAXIMUM-UDEGREE-IN-ROW()
1  maximum_row_uDegree = 0
2  for i = 1 to m
3      if maximum_row_uDegree > row_uDegree[i]
4          maximum_row_uDegree = row_uDegree[i]
5          row_uNode = i
```

Figure 4.9: Algorithm to find the row which has the maximum uDegree

Algorithm MAXIMUM-UDEGREE-IN-COLUMN() in Figure 4.10 calculates the column which has maximum uDegree. The algorithm is same as MAXIMUM-UDEGREE-IN-ROW(). *maximum_column_uDegree* stores maximum uDegree and *column_uNode* stores the column number.

```
MAXIMUM-UDEGREE-IN-COLUMN()
1  maximum_column_uDegree = 0
2  for i = 1 to n
3      if maximum_column_uDegree > column_uDegree[i]
4          maximum_column_uDegree = column_uDegree[i]
5          column_uNode = i
```

Figure 4.10: Algorithm to find the column which has the maximum uDegree

4.7 Heuristic approach

In this section we combine all the algorithms that we have discussed in the previous sections of this chapter and derive our heuristic algorithm from this. The first job in bidirectional partition of the matrix is to calculate the number of non-zeroes in every row and

column. Algorithm DEGREE-CALCULATION() in Figure 4.1 calculates degree of each row and column and stores it in *row_degree* and *column_degree*. Whenever a row or column is assigned to a group, we count the number of non-zeroes that are determined by that row or column. *nnz_count* calculates total number of non-zeroes that have so far been determined by the row and column groups. **while** loop in line 2 iterates until all non-zeroes of the matrix are determined . While making a new group we need to calculate distance-2 neighbors for every row and column. The list is necessary to determine uDegree of rows and columns. ROW-DISTANCE-2-NEIGHBORS-CALCULATION() in Figure 4.3 calculates the distance-2 neighbors of every row and COLUMN-DISTANCE-2-NEIGHBORS-CALCULATION() in Figure 4.4 finds the distance-2 neighbor list for every column. When a row is assigned to a group, the non-zeroes of that row are determined . If two columns have non-zeroes in that row, then they are not still distance-2 neighbor of one another if do not share non-zero in some other row. Similarly when a column is assigned to a group, the rows which have non-zeroes in that column, are not still a distance-2 neighbor. So we need to update the distance-2 neighbor lists whenever we start a new group.

MAXIMUM-DEGREE-CALCULATION() in line 5 calculates which row or column has maximum degree. If a row has maximum degree then the new group will consist of all rows, otherwise the group will consist of all columns. *direction* tells whether a row or column has maximum number of non-zeroes and *node* stores which row or column has that maximum non-zeroes. When we assign a row into a new group, the next step is to mark all the non-zeroes that are determined by that row. DATAUPDATEBASEDONROW(*node*) in Figure 4.5 updates data structures after a row is assigned to a group. Then we create a temporary group and put all the rows which are distance-2 neighbors of the currently computing group and calculate uDegree of each row. ROW-UDEGREECALCULATION(*node*) in Figure 4.7 updates uDegree of the rows after a row is assigned to a group. Then MAXIMUM-UDEGREE-IN -ROW() in Figure 4.9 determines the next row of the computing group. The row that has maximum connection with temporary group, will be the row in

```
HEURISTIC-APPROACH()
 1  DEGREE-CALCULATION()();
 2  while nnz_count ≠ nnz
 3      ROW-2PATH-LIST-CALCULATION()
 4      COLUMN-2PATH-LIST-CALCULATION();
 5      MAXIMUM-DEGREE-CALCULATION()()
 6      if direction == "row"
 7          add corresponding row into a new group
 8          ROW-uDEGREECALCULATION(node)
 9          DATAUPDATEBASEDONROW(node)
10          while true
11              MAXIMUM-uDEGREE-IN-ROW()
12              if maximum_row_uDegree == 0
13                  exit loop
14              add row_uNode into the same group
15              ROW-uDEGREECALCULATION(row_uNode)
16              DATAUPDATEBASEDONROW(row_uNode)
17      elseif direction == "column"
18          add corresponding column into a new group
19          COLUMN-uDEGREECALCULATION(node)
20          DATAUPDATEBASEDONCOLUMN(node)
21          while true
22              MAXIMUM-uDEGREE-IN-COLUMN()
23              if maximum_column_uDegree == 0
24                  exit loop
25              add column_uNode into the same group
26              COLUMN-uDEGREECALCULATION(column_uNode)
27              DATAUPDATEBASEDONCOLUMN(column_uNode)
```

Figure 4.11: Heuristic algorithm for bi-directional partitioning of matrices

the group. Every time we assign a row into a group, we need to mark the non-zeroes that are affected by that row. If there is no row whose uDegree is greater then 0 and all the non-zeroes are not determined than we have to make another group.

Similarly, when we assign a column to a new group then we have to mark the non-zeroes that are determined by that column by using the algorithm DATAUPDATEBASEDONCOLUMN(*node*) in Figure 4.6. For the other columns of the group we check uDegree of the columns and the node that has maximum uDegree will be the next column in the group. If there is no column whose uDegree is greater than 0, we create another group.

4.8 Verification of results

A row-column compression for a sparse matrix $A \in \Re^{m \times n}$ is a pair (V,W), $V \in \{0,1\}^{n \times p}$, $W \in \{0,1\}^{m \times q}$ such that $X = AV$ and $Y = W^T A$ can be computed, where in the matrix multiplication, scalar multiplication operation is replaced by logical AND operation. In performing logical AND, a non-zero scalar value is taken as true(1) and the value 0 is taken as false(0). Then matrices X and Y will be binary i.e. $0-1$ matrix.

Let I and J identify the (row,column) indices of the non-zeroes of matrix A that correspond to 1 in matrices X and Y, respectively.

4.8.1. Proposition. *Row-Column compression* (V,W) *completely determines the sparse matrix A if*

$$I \cup J = Index(nz(A))$$

where $Index(nz(A)$ *denotes the (row, column) entries of the non-zero entries of A.*

Proof. When AV computes X over $\{AND, +\}$, it checks the number of non-zeroes in each row that are affected by corresponding column group. The output of the operation for a row greater than 1, represents for that column group, the row contains more than one non-zero entries. As we know if there are more than one non-zero in a row for a column group then

the non-zeroes cannot be determined by that column group. If the output of the operation for a row is 1, then mark the non-zero and put its (row,column) position into set I.

Similarly, when we compute Y^T by calculating $W^T A$ over $\{AND, +\}$, we are checking non-zeroes in each column of A for a particular row group. If, for a particular column, the output is more than 1, then ignore the non-zeroes, because a row group will compute these non-zeroes. If the output for a column is 1, then mark the non-zero and put the index of that non-zero into J.

Now if the matrix A is appropriately partitioned into row groups and column groups, then every non-zeroes of the matrix must be determined by these groups. Set I stores all the non-zeroes that are determined by column groups and set J keeps all non-zeroes that are determined by row groups. Hence $I \cup J$ represents every non-zeroes of matrix A. □

Let us explain it with an example. Consider a sparse matrix:

$$A = \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 1 & 0 \end{bmatrix} \tag{4.2}$$

Here $Index(nz(A)) = \{a_{11}, a_{14}, a_{22}, a_{25}, a_{33}, a_{36}, a_{41}, a_{42}, a_{43}, a_{51}, a_{55}, a_{56}, a_{62}, a_{64}, a_{66}, a_{73}, a_{74}, a_{75}\}$. And the binary matrices representing the groups of the matrix are

$$V = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \text{ and } W^T = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 \end{bmatrix}$$

If we look at the binary matrices then we can see that the matrix is partitioned into groups where the members of the groups are $\{c_1, c_4\}$, $\{c_2, c_5\}$, $\{c_3, c_6\}$ and $\{r_1, r_2, r_3\}$. The rest of the rows remain un-grouped. While computing AV, rows of A are compared with each column group. While doing the operation $\{AND, +\}$ with the first row of A and the first column of V, the output of the operation is 2. So non-zeroes a_{11} and a_{41} cannot be determined by first column group. Now for the second and third row of A the output of the operation is 0. For fourth row of matrix A, the output of the operation is 1. So non-zero a_{41} can be determined by that column group and we put the non-zeroes into I. Similarly for rest of the rows of A, non-zero a_{51}, a_{64}, a_{74} are determined. In the same way we get non-zeroes a_{42}, a_{55}, a_{62}, a_{75} for second column and a_{43}, a_{56}, a_{66}, a_{73} for third column of V. So $I = \{a_{41}, a_{51}, a_{64}, a_{74}, a_{42}, a_{55}, a_{62}, a_{75}, a_{43}, a_{56}, a_{66}, a_{73}\}$.

Now while computing $W^T A$ to get Y^T, the only row group in W^T is computed with every column in A. From the operation $W^T A$ over $\{AND, +\}$, we get non-zeroes a_{11} , a_{22} , a_{33} , a_{14} ,a_{25}, a_{36}. Therefore $J = \{\ a_{11}$, a_{22} , a_{33} , a_{14} ,a_{25}, $a_{36}\}$. Now $I \cup J = \{a_{11}$, a_{14} , a_{22} , a_{25} , a_{33} ,a_{36} ,a_{41} , a_{42} , a_{43} ,a_{51} , a_{55} ,a_{56}, a_{62}, a_{64} , a_{66}, a_{73} , a_{74} , $a_{75}\}$ which is equal to $Index(nz(A))$. Hence our proposition is proved.

4.9 Conclusion

In this chapter we have presented a heuristic algorithm that partitions the non-zeroes of sparse Jacobian matrices bi-directionally to determine non-zeroes of the matrix. In the

algorithm, the first node is the node that has the maximum degree. The remaining nodes of the group are the nodes that have the maximum number of uDegree. Every time a node is selected we mark the non-zeroes that are in the node. The process continues until all non-zeroes are determined.

Chapter 5

Iterative Algorithm

In this chapter, we will discuss our iterative algorithm to partition the matrix bi-directionally to determine the non-zeroes of sparse Jacobian matrices. We have introduced a randomization technique to solve our iterative algorithm. In the following sections, we will discuss step by step the procedure to describe our coloring algorithm. Finally, we will combine all the procedures and build our iterative randomization algorithm.

5.1 Introduction

An iterative method approaches to the solution through a repetitive process. Every time, we use the outcomes of the previous iteration in the new iteration. One of the basic condition of path p-coloring is every 3-length path uses at least 3 colors. We derive our iterative algorithm mainly focusing this condition. At first, we extract all the 3-length paths from the bipartite graph. In the existing algorithms, the authors partition the matrix by selecting nodes using some criteria. Instead of selecting the nodes to color them, in this algorithm we assign random colors to all the nodes of the graph. Then in each iteration, we tried to figure out whether the color combination can make the graph p-colorable or not. If it cannot do it, then we change the color of a node in each iteration and then check whether the new color combination is sufficient to do that or not. To check whether a color combination in the nodes are sufficient to partition the matrix, we make a min-heap tree based on the number of colors in each path. Total colors of the path at root node of the min heap tree are 3 or more represent all the other paths in the tree contain at least 3 colors. So our goal is to make the total number of colors at the root node more than or equal to 3. If total colors at the root node is less than 3, then we randomly choose any row or column

node from the root node, change the color of that node so that total number of colors at the root node becomes 3 and delete that node from the heap tree along with all the paths that contain that row or column. Then we modify the total number of colors of the deleted node, reinsert the paths into heap tree again and check whether the total colors at the root node become 3 or not. If not, then continue the iteration until the total colors at root become 3 or more. If there is no result after a certain number of iterations, then we stop the iteration, increase the number of colors and restart the iteration.

5.2 Assign colors to the nodes of the graph

The first step of our iterative algorithm is to assign colors to the two sets of vertices (row set and column set). We assume a certain number of row and column colors and randomly color the nodes with the row and column colors.

```
ASSIGN-COLORS(row_colors,column_colors)
1  for i = 1 to m
2        row_node[i] = a random color among row_colors
3  for j = 1 to n
4        column_node[i] = a random color among column_colors
```

Figure 5.1: Algorithm to assign random colors to the nodes of a bipartite Graph

The algorithm ASSIGN-COLORS(*row_colors*,*column_colors*) in Figure 5.1 assigns colors to the nodes of the graph. *row_colors* is the number of colors allotted for rows and *column_colors* is total colors allotted for columns. Lines 1- 2 randomly assign colors to *row_node* and lines 3- 4 assign colors to *column_node*.

Let us consider the following sparse matrix :

$$A = \begin{bmatrix} \times & 0 & \times \\ 0 & \times & \times \\ \times & \times & \times \end{bmatrix} \tag{5.1}$$

Compressed row and compressed column structure of the matrix in Equaiton 5.1is given in Figure 5.2

col_ind	1	3	2	3	1	2	3

row_ptr	1	3	5	8

row_ind	1	3	2	3	1	2	3

col_ptr	1	3	5	8

Figure 5.2: Compressed row and compressed column structure of the matrix in Equation 5.1

The matrix in Equation 5.1 has 3 row nodes and 3 column nodes. Let us assume that total row colors needed to color the rows of the matrix is 2 and total column colors are also 2. Then r_1,r_2 and r_3 may get the colors rc_1, rc_2 and rc_1. Similarly c_1, c_2 and c_3 may get the colors cc_2, cc_1 and cc_2. Here rc_i is used to represent i-th row color and cc_j is used to represent j-th column color.

5.3 3-length path extraction

The next step in our iterative randomization algorithm is to extract all 3-length paths from the matrix. Each 3-length path contains four nodes, two of them are row nodes and the other two are column nodes. To find the nodes of a path from a matrix, we have to look into the compressed row and compressed column data structure of the matrix. From the

data structure first take a column index of any row. The column index is the first node of the path, and the row number is the second node of the path. The third node of the path is any other column indices of the row except the first node. The fourth node of the row is any of the row indices of the column in the third node except the second node.

From the data structure in Figure 5.2, we can see that r_1 has column indices c_1 and c_3. If we consider the first column index of r_1, then the first node of the path is c_1, the second node is the row number which is r_1, Third node is any other column indices of r_1 except c_1. If we take column index c_3, then the third node is c_3. The final node of the path is any row indices of c_3 except r_1. c_3 has three row indices which are r_1,r_2 and r_3. We cannot take r_1 as the fourth node, because r_1 has been used as second node . But we can take r_2 as the fourth node of the path. So one of the 3-length path of the matrix is $c_1 - r_1 - c_3 - r_2$. In the same way, the other paths that contain the first row are $c_1 - r_1 - c_3 - r_3$ and $c_1 - r_3 - c_1 - r_3$.

```
ALL-PATH-EXTRACTION()
 1  path_no = 0
 2  for i = 1 to m
 3       for k = row_ptr[i − 1] to row_ptr[i]-1
 4            second = i
 5            first = col_ind[k]
 6            for j = row_ptr[i − 1] to row_ptr[i]-1
 7                 if col_ind[j] ≠ first
 8                      third = col_ind[j]
 9                      for m = col_ptr[third−1] to col_ptr[third]-1
10                           fourth = row_ind[m]
11                           total_color = total colors in a path
12                           original_position = path_no
13                           current_position = path_no
14                           insert first, second,third,,fourth, total_color,original_position
                                    current_position into heap_tree_node
15                           path_no = path_no +1
```

Figure 5.3: Algorithm to extract all the 3-length paths of a matrix

The algorithm ALL-PATH-EXTRACTION() in Figure 5.3, extracts all the 3-length paths from a matrix. In the algorithm, variable *path_no* counts the number of paths in the matrix.

Variable *first* and *third* are used to store column nodes, *second* and *fourth* are used to store column nodes, and *total_color* is used to represent total colors in a path. In a single path we have two row nodes and two column nodes. According to p-path coloring rules, two adjacent nodes cannot have the same color. Therefore row colors and column colors are different. But in a path two row nodes or two column nodes can have same color. So if row colors and column colors are equal, then total number of colors in the path is 2. If row colors are different then total color increases by 1, and if column color varies then total color increases by 1. *original_position* and *current_position* are used to keep track of each path in the heap tree. *original_position* represents initial index of a path and *current_position* at index i represents the current index of a path in the tree which was previously at index i. Let us discuss the use of *original_position* and *current_position* with an example. At the beginning, *original_position* and *current_position* are initialized with the index number. So for all the paths of the matrix in Equation 5.1, the value of *original_position* and *current_position* are:

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13
$node_1$	c_1	c_1	c_3	c_2	c_2	c_3	c_1	c_1	c_1	c_2	c_2	c_2	c_3	c_3
$node_2$	r_1	r_1	r_1	r_2	r_2	r_2	r_3	r_3	r_3	r_3	r_3	r_3	r_3	r_3
$node_3$	c_3	c_3	c_1	c_3	c_3	c_2	c_2	c_3	c_3	c_1	c_3	c_3	c_1	c_2
$node_2$	r_2	r_3	r_3	r_1	r_3	r_3	r_2	r_2	r_1	r_1	r_1	r_2	r_1	r_2
originail_ position	0	1	2	3	4	5	6	7	8	9	10	11	12	13
current_ position	0	1	2	3	4	5	6	7	8	9	10	11	12	13

Figure 5.4: Configuration of nodes of the tree before starting an iteration

In Figure 5.4, *index* is used to represent path index number. $node_1$ and $node_3$ represent column nodes , and $node_2$ and $node_4$ are used to represent row nodes. *original_position* and *current_position* describes original position and current position of a path.

Let us assume the values in the heap tree become like in Figure 5.5 after some iteration.

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13
$node_1$	c_1	c_2	c_1	c_3	c_2	c_3	c_2	c_1	c_1	c_1	c_2	c_2	c_3	c_3
$node_2$	r_1	r_2	r_3	r_1	r_2	r_2	r_3	r_1	r_3	r_3	r_3	r_3	r_3	r_3
$node_3$	c_3	c_3	c_2	c_1	c_3	c_2	c_1	c_3	c_3	c_3	c_3	c_3	c_2	c_1
$node_2$	r_2	r_1	r_2	r_3	r_3	r_3	r_1	r_3	r_1	r_2	r_1	r_2	r_2	r_1
originail_ position	0	3	6	2	4	5	9	1	7	8	10	11	13	12
current_ position	0	7	3	1	4	5	2	8	9	6	10	11	13	12

Figure 5.5: Configuration of the nodes of the tree after some iterations of the algorithm

If we compare the two figures, Figure 5.4 and Figure 5.5, we can see that the value of *original_position* at index 1 of Figure 5.5 is 3. It means the path $c_2 - r_2 - c_3 - r_1$ which was initially at index 3 of Figure 5.4 is now at index 1. Similarly *current_position* at index 1 of Figure 5.5 is 7 means the path $c_1 - r_1 - c_3 - r_3$ which was initially at index 1 of Figure 5.4, is now at index 7 in Figure 5.5.

All of the above values are stored in a data structure named as *heap_tree_node*, which is used a node in the heap tree.

5.4 Heap tree construction

After extracting all the 3-length paths and calculating the total number of colors in each path, the next step is to make a heap tree based on the total number of colors of a path in the *heap_tree_node*. We mark each path with a number while we extract the paths using ALL-PATH-EXTRACTION(). *original_position* and *current_position* are used to keep track of the original position and the current position of every path in the heap tree.

The algorithm MAKE-HEAP() in Figure 5.6, inserts nodes into the heap tree based on the sequence of the paths given in Figure 5.4. While inserting *heap_tree_node* into heap tree, we compare the total number of colors of the path in the current *heap_tree_node* node with the total colors of the path in the parent node. If the total colors are less than or equal

```
MAKE-HEAP()
 1  for k = 1 to path_no
 2      child = k
 3      parent = k/2
 4      while total number of colors at the path in child is less or equal to total
            number of colors at the path in parent
 5          child_index = original_position value at index child
 6          parent_index = original_position value at index parent
 7          interchange row nodes, column nodes and color of the path between parent
                and child
 8          interchange value of original_position between parent and child
 9          set current_position value at index child_index to the value of index number
                of parent
10          set current_position value at index parent_index to the value of index number
                of child
11          child = parent
12          parent = parent/2
```

Figure 5.6: Algorithm to construct the heap tree

to the total colors of the path in parent node then interchange row nodes, column nodes, total colors and *original_position* between the two nodes. Every time we interchange the path information, we need to keep track of the index of their original path number which was in Figure 5.4. For that purpose, the algorithm in Figure 5.6 uses *child_index* to store *original_position* value of *child* index and *parent_index* to store *original_position* value of *parent* index. Then in line 9 and line 10, we change the *current_position* value at the index of *child_index* with the value in *parent*, and *current_position* value at the index of *parent_index* with the value of *child*.

Let us discuss the concept of *current_position* with an example. Suppose we want to interchange paths between the two indices 3 and 6 of Figure 5.5. *original_ position* value of path $c_3 - r_1 - c_1 - r_3$ at index 3 of Figure 5.5 is 2. So we change *current_position* at index 2 to 6. Similarly, *original_position* at index 6 is 9. So we change *current_position* value of index 9 to 3.

original_position value of path $c_2 - r_3 - c_1 - r_1$ at index 3 of Figure 5.7 is 9, which

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13
$node_1$	c_1	c_2	c_1	c_2	c_2	c_3	c_3	c_1	c_1	c_1	c_2	c_2	c_3	c_3
$node_2$	r_1	r_2	r_3	r_3	r_2	r_2	r_1	r_1	r_3	r_3	r_3	r_3	r_3	r_3
$node_3$	c_3	c_3	c_2	c_1	c_3	c_2	c_1	c_3	c_3	c_3	c_3	c_3	c_2	c_1
$node_2$	r_2	r_1	r_2	r_1	r_3	r_3	r_3	r_3	r_1	r_2	r_1	r_2	r_2	r_1
originail_ position	0	3	6	9	4	5	2	1	7	8	10	11	13	12
current_ position	0	7	6	1	4	5	2	8	9	3	10	11	13	12

Figure 5.7: Configuration of the structure of the tree if two nodes interchange their path in the node

means initially the path is at index 9 which we can see at Figure 5.4. *current_position* value at index 9 of Figure 5.7 is 3, which means the path $c_2 - r_3 - c_1 - r_1$ which was initially in index 9 is now at index 3. Similarly *original_position* of path $c_1 - r_3 - c_2 - r_2$ at index 9 is 8. *current_position* at index 8 is 9. So the path $c_1 - r_3 - c_2 - r_2$ was initially at index 8 is now at index 9.

5.5 Deletion of a node from heap tree

To delete a node from the heap tree, first swap the deleting node with the last node of the tree. Then compare the total number of colors at the path of the swapped node with the total number of colors at the path of its parent node. Continue swapping until total colors at child node is less than or equal to total colors of the path at parent node. If no swapping happens between child node and parent node then compare the total colors of the swapping node with the smaller total color of its children . Continue swapping until total color at parent node is greater then total colors of its children. The algorithm for deleting a node from the heap tree is given in Figure 5.8.

In algorithm POP-HEAP(), *path_number* tells the the path that we want to delete. At first, we check current position of *path_number*. *pop_index* stores the *current_position* of the path. *original_position* value of the last node is stored in *last_index*. Then we inter-

POP-HEAP(*path_number*)

```
1  pop_index = current_position value at index path_number
2  last_index = original_position value of last node
3  Interchange row nodes, column nodes of a path ,total path color and original_position
   between the node at index pop_index and last node
4  set current_position value of last node to pop_index and current_position value of
   path_number to the index of last node
5  child = pop_index
6  parent = child/2
7  while total number of colors at the path in child is less or equal to total
       number of colors at the path in parent
8      child_index = original_position value at index child
9      parent_index = original_position value at index parent
10     Interchange row nodes, column nodes of a path ,total path color and
       original_position between parent and child
11     set current_position value at index child_index to the value of index number
       of parent and current_position value at index parent_index to the value of child
12     child = parent
13     parent = parent/2
14 if there is no interchange
15     while total number of colors at the path in pop_index is greater than minimum of
       its child
16         Interchange row nodes, column nodes of a path ,total path color and
           original_position between parent and child
17         get the original_position of the parent and set current_position value of that
           index to the index of parent
18         get the original_position of the child node and set current_position value of that
           index to the index of child
```

Figure 5.8: Algorithm to delete a node from the heap tree

change row nodes, column nodes, total colors in the path between *pop_index* and *last_index*. After that we compare total colors of the path at *pop_index* with total colors of its parent node. While total colors at parent node is less or equal to total colors at child node, we continue interchanging values between two nodes. If no swapping has been done then we check whether the total colors at the path in *pop_index* is greater than any of its child nodes path color. When the total colors at the parent node is grater than the minimum of the path colors of its child node, we interchange values between the two nodes.

5.6 Inserting a node into heap tree

To insert a node into the heap tree, at first we insert the node as the last index of the tree. Then compare total colors of the last node with total colors of its parent node. When the total colors at the parent node is less than or equal to colors at child node, we interchange values between parent node and child node.

In the algorithm PUSH-HEAP(), first we find the last index of the tree and its parent index. *child* stores information of the path at the last index of heap tree and *parent* stores information of the parent node of *child*. When total colors at *child* is less than or equal to the total colors at *parent*, we interchange values between *child* and *parent*. After swapping, *child* gets the index of *parent* and *parent* gets the index of the parent of the *parent*. We continue swapping until total colors at *child* is less than total colors at *parent*.

```
PUSH-HEAP()
 1  child = the last index of the heap tree
 2  parent = child /2
 3  while total number of colors at the path in child is less or equal to total
          number of colors at the path in parent
 4      child_index = original_position value at index child
 5      parent_index = original_position value at index parent
 6      interchange row nodes, column nodes and color of the path between parent
          and child
 7      interchange value of original_position between parent and child
 8      set current_position value at index child_index to the value of index number
          of parent
 9      set current_position value at index parent_index to the value of index number
          of child
10      child = parent
11      parent = parent/2
```

Figure 5.9: Algorithm to insert a node into a heap tree

HEAP-TREE-TO-DETERMINE-PARTITION()

1 MAKE-HEAP()
2 **while** total colors of a path at root node is less than 3
3 Randomly chose a row or column node from the root node and recolor it among the assigned colors of that node type
4 Use POP-HEAP(*path_number*) where *path_number* is the path number of the path at root node
5 Insert the node into heap tree using PUSH-HEAP()
6 Find all the path that contains the node which color has been modified
7 **for** each path update total color numbers of the path , delete the path using POP-HEAP() and then again reinsert the path using PUSH-HEAP()
8 **if** path colors at root node becomes 3 or more then **return** true
9 **if** the number of iteration is more than 20 times of the total path in the matrix then exit the loop and **return** false

Figure 5.10: Algorithm to determine whether the specific row and column color numbers can determine the non-zeroes of a matrix or not

5.7 Use of heap tree to determine partitioning

In this section we will discuss whether a certain number of row and column colors are sufficient to color the matrix. If the total colors of a path at the root node is equal to or more than 3, then every path has at least 3 colors. Hence path p-coloring of the matrix with the color combination is true. The algorithm to determine whether a color distribution satisfies the conditions of path p-coloring or not is given in Figure 5.10.

The first step to determine whether a particular number of row colors and column colors are sufficient to determine the matrix is to construct a heap tree using the algorithm MAKE-HEAP(). MAKE-HEAP() builds a min heap tree based on a total number of colors. Then we start to check whether the total number of colors at root node is less than 3 or not. If it is less than 3, then randomly chose a node from the path at the root node and assign another color to that node so that path color at root node becomes 3. Now delete the path from heap tree using POP-HEAP() and reinsert it using PUSH-HEAP() algorithm. The next step is to find all the paths that contain the randomly selected node, recalculate the total color of the paths, delete the paths from the tree using POP-HEAP() and again

reinsert it into the tree using PUSH-HEAP(). In this way, if the total colors at the root node is greater or equal to 3, then we can say that the colors are sufficient to partition the matrix. If the total number of iterations is more than 20 times the total number of 3-length paths of the bipartite graph, then we quit the iteration for that color numbers and increase the color numbers.

5.8 Finding the optimized result

Every time we determine row and column colors, our first job is to assign the colors randomly among the row and column node. Then we extract all the 3-length paths from the graph using ALL-PATH-EXTRACTION() algorithm. After that we call the HEAP-TREE-TO-DETERMINE-PARTITION() to determine whether the colors are sufficient to determine the non-zeroes of the matrix or not. To get the optimized result, first we fix the row color and try to find how many column colors are required to determine non-zeroes of the matrix. We used row color from 1 to ρ_{max}, where ρ_{max} is the maximum number of non-zeroes in a row. Then we again check our HEAP-TREE-TO-DETERMINE-PARTITION() by varying column color from 1 to maximum number of non-zeroes in a column and try to find how many row colors are sufficient to partition the matrix to determine the non-zeroes. Among the color combinations, the minimum color combination will be the optimal result.

5.9 Conclusion

In this chapter, we tried to partition the matrix by randomly coloring the vertices with a certain number of colors. To the best of our knowledge, this is the first time the use of randomization to determine the non-zeroes to partition the matrix has been introduced. Our technique provides good results, but it takes a lot of time.

Chapter 6

Experimental Results

In this chapter, we present computational results of the algorithms proposed in this thesis. In Section 6.1 we give details of test data sets that we have used in our experiments. Lower bound results are discussed in Section 6.2. In Section 6.3, we provide our heuristic results and compare them with other methods. In Section 6.4, we compare our lower bound and heuristic with the lower bound and heuristic of [21]. Section 6.5 describes the numerical experiments with our iterative algorithm. Finally in Section 6.6 we conclude the chapter.

6.1 Test matrices

In this section, we discuss the details of our data sets. The data set in Table 6.1 is for the large matrices which are collected from University of Florida sparse matrix collection [3] and Matrix Market Collection [2]. These matrices are used to compare our algorithms with unidirectional partitioning results of [14]. The data set in Table 6.2 is obtained from Harwell-Boeing test matrices [1] and University of Florida Matrix Collection [3]. Part of these data sets are used to compare our lower bound with the lower bound of Juedes and Jones[21].

Table 6.1: Matrix statistics for set 1

Matrix	**n**	**m**	**nnz**	ρ_{max}	ρ_{min}	κ_{max}	κ_{min}
af23560	23560	23560	484256	21	11	21	10
cage11	39082	39082	559722	31	3	31	3
cage12	130228	130228	2032536	33	5	33	5

Continued on next page

Table 6.1 – continued from previous page

Matrix	**n**	**m**	**nnz**	ρ_{max}	ρ_{min}	κ_{max}	κ_{min}
e30r2000	9661	9661	306356	62	8	62	8
e40r0100	17281	17281	553956	62	8	62	8
lhr10	10672	10672	232633	63	1	36	1
lhr14	14270	14270	307858	63	1	36	1
lhr34	35152	35152	764014	63	1	36	1
lhr71c	70304	70304	1528092	63	1	36	1
lpcrea	3516	7248	18168	360	1	14	1
lpcreb	9648	77137	260785	844	1	14	1
lpcred	8926	73948	246614	808	1	13	1
lpfit2d	25	10524	129042	10500	1427	17	1
lpdfl001	6071	12230	35632	228	2	14	1
lpken11	14694	21349	49058	122	1	3	1
lpken13	28632	42659	97246	170	1	3	1
lpken18	105127	154699	358171	325	1	3	1
lpmarosr7	3136	9408	144848	48	5	46	1
lppds10	16558	49932	107605	96	1	3	1
lppds20	33874	108175	232647	96	1	3	1
lpstocfor3	16675	23541	76473	15	1	18	1

Table 6.2: Matrix statistics for set 2

Matrix	**n**	**m**	**nnz**	ρ_{max}	ρ_{min}	κ_{max}	κ_{min}
abb313	313	176	1557	6	1	26	2
adlittle	56	138	424	27	1	11	1
agg	488	615	2862	19	2	43	1
agg2	516	758	4740	49	2	43	1
agg3	516	758	4756	49	2	43	1
arc130	130	130	1282	124	1	124	1
ash219	219	85	438	2	2	9	2
ash292	292	292	2208	14	4	14	4
ash331	331	104	662	2	2	12	3
ash608	608	188	1216	2	2	12	2
ash958	958	292	1916	2	2	13	3
blend	74	114	522	29	2	16	1
bore3d	233	334	1448	73	1	28	1
bp0	822	822	3276	266	1	20	1
bp1000	822	822	4661	308	1	21	1
bp1200	822	822	4726	311	1	21	1
bp1400	822	822	4790	311	1	21	1
bp1600	822	822	4841	304	1	21	1
bp200	822	822	3802	283	1	21	1
bp400	822	822	4028	295	1	21	1
bp600	822	822	4172	302	1	21	1

Continued on next page

Table 6.2 – continued from previous page

Matrix	**n**	**m**	**nnz**	ρ_{max}	ρ_{min}	κ_{max}	κ_{min}
bp800	822	822	4534	304	1	21	1
can1054	1054	1054	12196	35	6	35	6
can1072	1072	1072	12444	35	6	35	6
can256	256	256	2916	83	4	83	4
can268	268	268	3082	37	4	37	4
can292	292	292	2540	35	4	35	4
can634	634	634	7228	28	2	28	2
can715	715	715	6665	105	2	105	2
curtis54	54	54	291	12	3	16	3
dwt1007	1007	1007	8575	10	3	10	3
dwt1242	1242	1242	10426	12	2	12	2
dwt2680	2680	2680	25026	19	4	19	4
dwt419	419	419	3563	13	13	6	6
dwt59	59	59	267	6	2	6	2
eris1176	1176	1176	18552	99	2	99	2
fs5411	541	541	4285	11	1	541	5
fs5412	541	541	4285	11	1	541	5
gent113	113	113	655	20	1	27	1
ibm32	32	32	126	8	2	7	2
impcola	207	207	572	8	1	5	1
impcolb	59	59	312	7	2	12	1
impcolc	137	137	411	8	1	8	1

Continued on next page

Table 6.2 – continued from previous page

Matrix	**n**	**m**	**nnz**	ρ_{max}	ρ_{min}	κ_{max}	κ_{min}
impcold	425	425	1339	10	1	10	1
impcole	225	225	1308	12	1	23	1
israel	174	316	2443	119	2	136	1
lunda	147	147	2449	21	5	21	5
lundb	147	147	2441	21	5	21	5
scagr25	471	671	1725	10	1	9	1
scagr7	129	185	465	10	1	9	1
shl0	663	663	1687	422	1	4	1
shl200	663	663	1726	440	1	4	1
shl400	663	663	1712	426	1	4	1
stair	356	614	4003	36	2	34	1
standata	359	1274	3230	745	2	10	1
str0	363	363	2454	34	1	34	1
str200	363	363	3068	30	1	26	1
str400	363	363	3157	33	1	34	1
tuff	333	628	4561	113	0	25	1
vtpbase	198	346	1051	38	1	12	1
watt2	1856	1856	11550	128	1	65	2
west0067	67	67	294	6	1	10	2
west0381	381	381	2157	25	1	50	1
west0497	497	497	1727	28	1	55	1
will199	199	199	701	6	1	9	2

Continued on next page

Table 6.2 – continued from previous page

Matrix	**n**	**m**	**nnz**	ρ_{max}	ρ_{min}	κ_{max}	κ_{min}
will57	57	57	281	11	2	11	2

In the tables, Table 6.1 and Table 6.2, under the column labelled **Matrix**, we provide the name of the matrix, and the columns labelled **n**, **m** and **nnz** are used to represent the number of columns, rows, and total number of non-zeroes in the matrix respectively, ρ_{max} and ρ_{min} represent maximum and minimum number of non-zeroes in any row of the matrix and κ_{max} and κ_{min} represent maximum and minimum number of non-zeroes in any column of the matrix.

6.2 Lower bound comparison

In **DSM** [4], a column intersection graph is used to represent the matrix. A unidirectional partitioning of the columns into structurally orthogonal groups is obtained by coloring the column intersection graph. A greedy coloring heuristic is employed to assign colors to columns. A well known easily computable lower bound on the number of groups in a unidirectional partitioning is given by the maximum of non-zeroes in any row of the Jacobian matrix. However ρ_{max} does not constitute a lower bound for bi-directional partitioning. To see this, consider the "arrow-head" example of [16].

$$A = \begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & & & \\ \times & & \times & & \\ \times & & & \times & \\ \times & & & & \times \end{bmatrix}$$

Figure 6.1: An arrow-head example of a sparse matrix

Matrix $A \in \Re^{m \times n}$ in Figure 6.1 has an "arrow-head" sparsity pattern. For matrix A,

$\rho_{max} = n$. As discussed in [16], clearly the non-zero entries of matrix A can be determined with 3 groups using bi-directional partitioning.

6.3 Heuristic results

In this section, we will compare our bi-directional heuristic partitioning result with unidirectional **RLF** ordering provided by [14]. Hasan [14], in his thesis, assigns colors using a column intersection graph. For comparison purposes, we used **DSJM** [14] code with row intersection graph. In Table 6.3 we make a comparison of our heuristic with the unidirectional **RLF** algorithm.

Table 6.3: Heristic result comparison for dataset 1

Matrix	**HRCl**	**HCCl**	**HTCl**	**RRCl**	**RCCl**
af23560	0	34	**34**	37	37
cage11	0	55	55	**54**	**54**
cage12	0	58	58	**56**	**56**
e30r2000	0	66	66	66	**65**
e40r0100	0	66	**66**	**66**	67
lhr10	33	7	**40**	62	64
lhr14	34	6	**40**	63	63
lhr34	35	5	**40**	63	63
lhr71c	35	5	**40**	64	63
lpcrea	14	0	**14**	**14**	360
lpcreb	15	0	**15**	**15**	844
lpcred	14	0	14	**13**	808
lpfit2d	0	21	**21**	**21**	10500

Continued on next page

Table 6.3 – continued from previous page

Matrix	HRCl	HCCl	HTCl	RRCl	RCCl
lpdfl001	0	14	**14**	**14**	228
lpken11	0	4	**4**	**4**	122
lpken13	0	4	**4**	**4**	170
lpken18	0	4	**4**	**4**	325
lpmarosr7	0	78	78	**76**	96
lppds10	0	5	5	**4**	96
lppds20	0	5	5	**4**	96
lpstocfor3	0	15	**15**	18	**15**

In the tables, Table 6.3 and Table 6.4, **HRCl** and **HCCl** represent the row colors and the column colors required using our heuristic algorithm. **HTCl** is the sum of **HRCl** and **HCCl** which is used to represent total colors required by our algorithm. **RRCl** and **RCCl** represent colors required by row intersection graph and column intersection graph, respectively by using Hasan's [14] **RLF** ordering algorithm.

In the Table 6.4, for the data set in Table 6.2, we have added another column **MBCl** to Table 6.4 which represents Bi-directional coloring results developed by Goyal[11] along with all the columns in Table 6.3.

Table 6.4: Heristic result comparison for dataset2

Matrix	HRCl	HCCl	HTCl	RRCl	RCCl	MBCl
abb313	0	10	**10**	26	**10**	**10**
adlittle	12	0	12	12	27	**11**

Continued on next page

Table 6.4 – continued from previous page

Matrix	HRCl	HCCl	HTCl	RRCl	RCCl	MBCl
agg	0	20	20	43	**19**	21
agg2	2	25	**27**	43	49	31
agg3	2	26	**28**	43	49	29
arc130	9	17	**26**	124	124	**26**
ash219	0	4	**4**	**4**	**4**	5
ash292	9	0	**9**	**9**	14	13
ash331	0	6	**6**	12	**6**	**6**
ash608	0	6	**6**	12	**6**	**6**
ash958	0	6	**6**	13	**6**	**6**
blend	6	10	**16**	**16**	29	17
bore3d	22	3	**25**	28	73	**25**
bp0	14	3	17	20	266	**16**
bp1000	22	0	22	**21**	308	**21**
bp1200	21	0	21	**21**	311	**21**
bp1400	21	0	**21**	**21**	311	**21**
bp1600	21	0	**21**	**21**	304	**21**
bp200	15	4	19	21	283	**17**
bp400	16	4	**20**	21	295	**20**
bp600	17	2	**19**	21	302	21
bp800	22	0	22	**21**	304	**21**
can1054	0	32	32	35	35	**30**
can1072	0	32	32	35	35	**31**

Continued on next page

Table 6.4 – continued from previous page

Matrix	HRCl	HCCl	HTCl	RRCl	RCCl	MBCl
can256	25	3	**28**	83	83	29
can268	10	22	32	37	37	**30**
can292	3	14	**17**	35	35	19
can634	5	21	**26**	28	28	29
can715	3	16	**19**	105	105	21
curtis54	1	10	**11**	16	12	12
dwt1007	0	10	**10**	**10**	**10**	11
dwt1242	0	13	**13**	**13**	**13**	15
dwt2680	2	17	**19**	**19**	**19**	21
dwt419	0	15	**15**	**15**	**15**	16
dwt59	0	6	**6**	**6**	**6**	7
eris1176	7	85	**92**	99	99	93
fs5411	0	13	**13**	541	12	14
fs5412	0	13	**13**	541	12	14
gent113	6	12	**18**	27	20	19
ibm32	1	6	**7**	**7**	8	8
impcola	5	0	**5**	**5**	8	6
impcolb	0	10	**10**	16	**10**	11
impcolc	0	8	8	8	8	**6**
impcold	0	10	10	11	10	**6**
impcole	0	21	**21**	31	**21**	22
israel	41	7	**48**	136	119	50

Continued on next page

Table 6.4 – continued from previous page

Matrix	HRCl	HCCl	HTCl	RRCl	RCCl	MBCl
lunda	0	23	23	23	**22**	26
lundb	0	23	**23**	**23**	**23**	26
scagr25	9	0	9	9	10	**8**
scagr7	9	0	9	9	10	**8**
shl0	4	0	**4**	**4**	422	**4**
shl200	4	0	**4**	**4**	440	**4**
shl400	4	0	**4**	**4**	426	**4**
stair	36	0	**36**	**36**	**36**	**36**
standata	10	0	10	10	745	**9**
str0	5	22	27	35	34	**26**
str200	31	0	31	31	**30**	**30**
str400	17	19	36	36	36	**33**
tuff	11	8	**19**	25	114	20
vtpbase	2	11	13	13	18	**12**
watt2	1	11	**12**	65	128	13
west0067	0	8	**8**	12	**8**	10
west0381	6	4	**10**	50	28	12
west0497	3	14	17	55	28	**16**
will199	2	7	9	69	**7**	8
will57	0	7	**7**	11	11	11

From the Table 6.3 we can see that for the matrix *lffit2d*, total color required by the column intersection graph is 10500, but total color required by the row intersection graph and our heuristic algorithm is only 21. This is because there are dense rows in the matrix *lffit2d*. For the matrix *lhr34*, there are both dense rows and dense columns in the matrix, Thus the total color needed to determine the matrix by both the column intersection graph and the row intersection graph is 63, while it requires only 40 colors by our heuristic algorithm. We make the following observations from the Table 6.3:

- Out of 25 problems, **HTCl** achieves the best coloring on 14 of them.
- On 17 problems bi-directional partitioning results are better than the best unidirectional partitioning.
- On problems where **HTCl** achieves strictly better partitioning, it performs significantly better(see *lhr* examples).

Total colors required by the row intersection graph using **RLF** for all matrices in Table 6.3 is 722. But for our heuristic algorithm it is only 632. For the matrices in Table 6.4, total color required by the row intersection graph is 2956 and total color required by column intersection graph is 6436. Bidirectional partitioning results of [11] requires 1221 colors. But our algorithm requires a total of 1193 colors for the matrices.

6.4 Comparison with ASBC

In Table 6.5, we compared our best lower bound and heuristic partitioning results with that of [21]. Here LB_{ASBC} represents lower bound result of [21] and **LB** represents our best lower bound result. Summing the lower bounds over the set of test matrices, we find that the lower bound proposed in this thesis (total groups = 366) is better than that of **ASBC**(total groups = 284). With respect to partitioning, the number of colors required to partition the matrices by using [21] are 800, while our heuristic algorithm requires only 744 colors.

So it can be said that, on the test problems considered, our algorithms (Lower bound and Heuristic partition) give better results than [21].

Table 6.5: Lower bound and color comparison with ASBC

Matrix	LB_{ASBC}	**LB**	$COLOR_{ASBC}$	**COLOR**
watt2	7	7	16	**12**
cannes256	12	12	27	28
cannes292	9	9	22	**17**
cannes634	12	12	29	**26**
cannes715	10	10	23	**19**
cannes1054	12	12	30	35
cannes1072	12	12	31	35
impcolc	3	**4**	7	8
impcold	4	4	7	10
impcolde	6	**10**	21	21
west0067	5	5	9	**8**
west0381	6	6	12	**10**
west0497	4	**7**	16	17
gent113	6	**9**	18	18
arc130	10	**16**	26	26
abb313	5	6	17	10
ash219	2	2	8	**4**
ash292	8	8	19	**14**
ash331	2	2	10	**6**

Continued on next page

Table 6.5 – continued from previous page

Matrix	LB_{ASBC}	**LB**	$COLOR_{ASBC}$	**COLOR**
ash608	2	2	11	**6**
ash958	2	2	12	**6**
bp0	4	**7**	16	17
bp200	5	**7**	17	19
bp400	5	**7**	19	20
bp800	6	**8**	22	22
bp1000	6	**8**	22	22
curtis54	6	6	12	**11**
eris1176	16	**80**	92	92
fs5411	8	8	18	**13**
fs5412	8	8	18	**13**
ibm32	4	4	9	**8**
lunda	17	**18**	26	**24**
lundb	17	**18**	27	**23**
shl0	3	3	7	**4**
shl200	3	3	7	**4**
shl400	3	3	6	**4**
str0	7	**17**	25	**27**
str200	9	**17**	31	32
str400	9	**18**	36	36
will57	5	**6**	10	10
will199	4	4	9	**7**

6.5 Iterative results

We have tested our iterative algorithm by varying row colors from 1 to ρ_{max}. We perform the same algorithmic steps in terms of columns as well.

Table 6.6: Iterative results

Matrix	**m**	**n**	**nnz**	**NOP**	ρ_{max}	κ_{max}	**RColor**	**CColor**
imb32	32	32	126	1478	8	7	1	8
ash331	331	104	662	3854	2	12	1	6
cannes24	24	24	92	1481	8	9	1	8
will57	57	57	281	6770	11	11	1	11
will199	57	57	281	6105	6	9	1	8
ash608	608	188	1216	7236	2	12	1	6
curtis54	54	54	291	8002	12	16	1	12
g10	100	100	460	6280	5	5	1	8
ash958	958	292	1916	11548	2	13	1	6

Table 6.5 depicts test results for a small subset of matrices. Here **m**, **n** and **nnz** represent the total number of rows, columns and number of non-zeroes of the corresponding matrix. **NOP** indicates number of 3-paths($P3$) in the matrix. Maximum number of non-zeroes in a row and maximum number of non-zeroes in a column are represented by ρ_{max} and κ_{max}. **RColor** and **CColor** indicate the total number of row colors and column colors required to determine all the non-zeroes of a matrix. From the Table 6.5, we can see that as the matrix size increases, the total number of 3-paths also increases. As a result of the increase in $P3$

paths, the time to partition the matrices also increases.

6.6 Conclusion

In this chapter, we have provided our lower bound and heuristic test results and compared our results with other algorithms from the literature. In our heuristic algorithm, we are mainly concentrated with the minimization of the number of groups of columns and rows, such that all the non-zeroes are uniquely determined. For our iterative algorithm, we have introduced a randomization technique in a matrix partitioning problem, which was never done before to the best of our knowledge.

Chapter 7

Conclusion and future work

In this thesis, we have proposed a new lower bound on the number of groups in a bi-directional partition of sparse Jacobian matrices. Our algorithms generalize the approach of [15] and [21]. In our algorithms, we permute the rows and the columns so that non-zeroes of the matrix that form a dense sub-matrices come close to one another. We then apply the lower bound expression of [15, 21] in every square sub-matrices. The largest value over all these sub-matrices is a lower bound for the matrix. We have used two types of permutation. One is based on the number of non-zeroes in the rows and the columns, the other is based on the position of the non-zeroes in rows and columns. We also developed a heuristic algorithm to partition the matrix into groups of rows and columns. We have used randomization to color the nodes of the bipartite graph that we derive from the sparse matrix. We have provided extensive numeric testings, and our test results show that on a standard set of test problems our heuristic algorithm performs better than existing algorithms. We have implemented our algorithms using C++.

7.1 Future research direction

For future work, we would like to give the following suggestions:

- The first element of every group in our heuristic algorithm is the row or the column that has the maximum number of non-determined non-zeroes. Instead of doing that, we may insert a collection of nodes that determine the maximum number of non-determined non-zeroes. This new strategy may reduce the size of the bipartition.

- In our heuristic algorithm, we tried to minimize the number of groups required to

partition the matrix. Time complexity of our algorithm is slightly higher than the existing algorithms, but our algorithm provides a better result. So in the future development of this problem, we should focus on efficient implementation.

- While doing our iterative algorithm, we chose a node from the path at root node by random selection. An alternative is to use different value ordering and variable ordering strategies [23, 24] to select nodes to recolor then we might get a better result.

Bibliography

[1] URL: http://math.nist.gov/MatrixMarket/collections/hb.html. [Online; accessed May-2015].

[2] URL: http://math.nist.gov/MatrixMarket/matrices.html. [Online; accessed May-2015].

[3] URL: http://www.cise.ufl.edu/research/sparse/matrices/. [Online; accessed May-2015].

[4] Thomas F Coleman, Burton S Garbow, and Jorge J Moré. Software for estimating sparse Jacobian matrices. *ACM Transactions on Mathematical Software (TOMS)*, 10(3):329–345, 1984.

[5] Thomas F Coleman and Jorge J Moré. Estimation of sparse Jacobian matrices and graph coloring blems. *SIAM journal on Numerical Analysis*, 20(1):187–209, 1983.

[6] Thomas F Coleman and Jorge J More. Estimation of sparse Hessian matrices and graph coloring problems. *Mathematical Programming*, 28(3):243–270, 1984.

[7] Thomas F Coleman and Arun Verma. The efficient computation of sparse Jacobian matrices using automatic differentiation. *SIAM Journal on Scientific Computing*, 19(4):1210–1233, 1998.

[8] AR Curtis, Michael JD Powell, and John K Reid. On the estimation of sparse Jacobian matrices. *J. Inst. Math. Appl*, 13(1):117–120, 1974.

[9] Michael R Garey and David S Johnson. Computers and intractability: a guide to the theory of NP-completeness. 1979. *San Francisco, LA: Freeman*, 1979.

[10] Assefaw Hadish Gebremedhin, Fredrik Manne, and Alex Pothen. What color is your Jacobian? Graph coloring for computing derivatives. *SIAM review*, 47(4):629–705, 2005.

[11] M. Goyal. Graph coloring in sparse derivative matrix computation. 2005. [M.Sc Thesis].

[12] Mini Goyal and Shahadat Hossain. Bi-directional determination of sparse Jacobian matrices: approaches and algorithms. *Electronic Notes in Discrete Mathematics*, 25:73–80, 2006.

[13] Andreas Griewank and Andrea Walther. *Evaluating derivatives: principles and techniques of algorithmic differentiation*. SIAM, 2008.

[14] Mahmudul Hasan. Dsjm: a software toolkit for direct determination of sparse Jacobian matrices. 2011. [M.Sc Thesis].

[15] AKM Shahadat Hossain. *On the computation of sparse Jacobian matrices and Newton steps*. PhD thesis, 1998.

[16] AKM Shahadat Hossain and Trond Steihaug. Computing a sparse Jacobian matrix by rows and columns. *Optimization Methods and Software*, 10(1):33–48, 1998.

[17] Shahadat Hossain and Trond Steihaug. Reducing the number of ad passes for computing a sparse Jacobian matrix. In *Automatic differentiation of algorithms*, pages 263–270. Springer, 2002.

[18] Shahadat Hossain and Trond Steihaug. Graph coloring in the estimation of sparse derivative matrices: Instances and applications. *Discrete Applied Mathematics*, 156(2):280–288, 2008.

[19] Shahadat Hossain and Trond Steihaug. Graph models and their efficient implementation for sparse Jacobian matrix determination. *Discrete Applied Mathematics*, 161(12):1747–1754, 2013.

[20] Shahadat Hossain and Zhenshuan Zhang. Cseggraph: Column segment graph generator. *University of Lethbridge*, 4401, 2003.

[21] David Juedes and Jeffrey Jones. Coloring Jacobians revisited: a new algorithm for star and acyclic bicoloring. *Optimization Methods and Software*, 27(2):295–309, 2012.

[22] Anany V Levitin. *Introduction to Design & Analysis of Algorithms: For Anna University, 2/e*. Pearson Education, 2009.

[23] Patrick Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational intelligence*, 9(3):268–299, 1993.

[24] Barbara M Smith. The brélaz heuristic and optimal static orderings. In *Principles and Practice of Constraint Programming–CP99*, pages 405–418. Springer, 1999.