STAR BI-COLORING OF BIPARTITE GRAPHS

AHMED SHOEB AL HASAN
Bachelor of Science, Military Institute of Science and Technology, 2010

A thesis submitted
in partial fulfilment of the requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

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

STAR BI-COLORING OF BIPARTITE GRAPHS

AHMED SHOEB AL HASAN

Date of Defence: December 09, 2019

Dr. Shahadat Hossain Thesis Supervisor	Professor	Ph.D.
Dr. Robert Benkoczi Thesis Examination Committee Member	Associate Professor	Ph.D.
Dr. Saurya Das Thesis Examination Committee Member	Professor	Ph.D.
Dr. Howard Cheng Chair, Thesis Examination Committee	Associate Professor	Ph.D.

Dedication

Dedicated to my parents, wife and daughter.

Abstract

Evaluation of the Jacobian is the most computationally expensive operation while solving a non-linear system. Knowledge of the sparsity pattern in advance reduces the computational cost. Bi-directional partitioning to determine non-zeroes in the sparse matrix works better than unidirectional partitioning for dense rows and dense columns. We have developed a bidirectional coloring algorithm that determines all the non-zeroes of a sparse Jacobian matrix. Our algorithm is inspired by complete direct cover [17]. Several numerical experiments have been carried out on standard data sets. Test results ensure that our proposed algorithm works better than existing algorithms. We have implemented our algorithm using the data structures and partitioning algorithms defined in software tool kit DSJM (Determine Sparse Jacobian Matrices). We have added new procedures in DSJM, which facilitates bi-directional partitioning.

Acknowledgments

I would like to express my deepest gratitude to my supervisor Dr. Shahadat Hossain. Without his guidance and encouragement, it would not be possible for me to finish the thesis work. Dr. Hossain offered his valuable suggestions and recommendations throughout the master's program. Thank you, sir, for your continued support and encouragement towards me.

I would like to express my sincere appreciation towards my supervisory committee members, Dr. Robert Benkoczi and Dr. Saurya Das. I am really thankful for their guidance and feedback.

My family encouraged me a lot throughout my master's program. I am very grateful to my parents for their unconditional love and support. I would also like to thank my in-laws as well as my brothers and sister for their continuous support.

I would like to express my love and gratitude to my wife (Nazia) and daughter (Tasmia) for being with me and support me. Nazia and Tasmia, without you two, it would not be possible for me to go through this journey.

I am grateful to the School of Graduate Studies for providing financial assistance for my graduate study and research work.

I would also like to thank my friends and well-wishers.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

A set of vertices of a graph grouped into two disjoint sets such that no two vertices from the same set are connected is known as a bipartite graph. A bipartite graph is a k-partite graph where k=2.

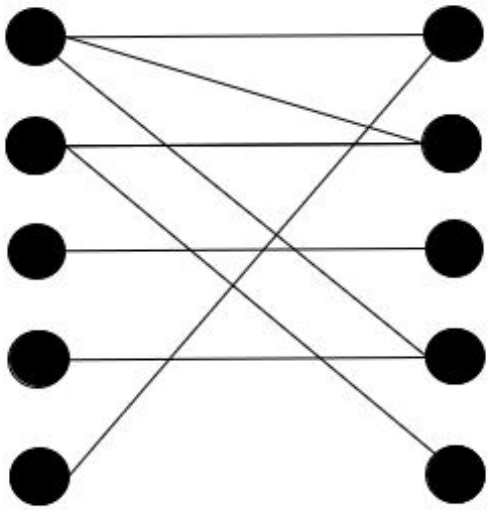

Figure 1.1: Bipartite graph

When the vertices of a graph are colored in a way that no two adjacent vertices have the same color and any path of length 3 is not bi-colored, then it is called star bi-coloring [7]. It has been established that the star bi-coloring problem is computationally hard (NP-Hard). Consequently, a large segment of research endeavors has been directed toward designing efficient heuristics that run faster and achieve reasonably good solutions. Unfortunately, the quality of solutions returned by heuristics is challenging to assess. A good lower bound on

the chromatic number (number of colors in an optimum coloring) is helpful in this regard. A good lower bound on the chromatic number in a standard graph coloring is the size of a (maximum) clique. For star bi-coloring, a similar structural measure is much more challenging to characterize.

1.1 Problem Definition

Consider a bipartite graph G = (V_c, V_r, E) where V_c and V_r are sets of vertices in the bipartition and the edges E connect vertices in V_r and V_c. A star bi-coloring of graph G is to assign colors (integer labels) to the vertices such that

1. vertices connected by an edge, assume distinct colors.

2. every path of four vertices assumes at least three distinct colors.

The objective is to assign colors such that the number of colors is minimized. The star bi-coloring problems arise, among others, in the determination of sparse derivative matrices in nonlinear optimization [5, 10, 19]. In this research, we plan to approach the determination of non-zero entries of the sparse matrix as the star bi-coloring of the bipartite graph. All the rows of a sparse matrix are considered as one disjoint set of vertices, and all the columns of that matrix are considered as another disjoint set of vertices of the bipartite graph. The non-zero entries of the sparse matrices need to be determined for the evaluation of the Jacobian matrix. Our proposed star bi-coloring algorithm will determine all the non-zero entries using bi-directional partitioning. Bidirectional partitioning for determining non-zeroes in a dense sparse matrix works better than uni-directional partitioning [13]. The partitioning will start from the vertex with maximum degree. A vertex will be grouped together with the vertices from the same disjoint set, and vertices in the same group can not have a path of length 2. This approach is based on complete direct cover [17]. All the vertices of the bipartite graph will be grouped in numeral groups by determining all the non-zeroes of the sparse matrix.

1.2 Objective

- Finding out row-column compression [9] for Jacobian's.
- Developing methods to determine non-zeroes of large sparse Jacobian matrices.
- Implementing star bi-coloring for the bipartite graph in DSJM (Determine Sparse Jacobian Matrices) [16] using complete direct cover [17].
- Implementing star bi-coloring of the bipartite graph for different ordering and partition algorithms and compare the results with the natural order.

1.3 Our Contribution

Our contributions of this thesis are as follows-

1. In-depth study of software tool kit DSJM (Determine Sparse Jacobian Matrices) [16]. The data structures and the algorithms for ordering and partitioning included in DSJM are studied thoroughly.
2. Implementation of finding the degree of vertices of a bipartite graph using efficient data structures used in DSJM.
3. Implementation of finding the transpose of a matrix using the data structures used in DSJM.
4. Implementation of ordering and partitioning algorithms for row intersection graph of DSJM using bucket heap data structure.
5. Implementation of a star bi-coloring algorithm using efficient data structures used in DSJM.
6. Numerical experiments to show the differences between new implementation and the previous ones.

1.4 Thesis Organization

There are six chapters in the thesis. Chapter 1 introduces the research work with problem definition. Then we discuss the objectives of this works along with the contributions made during this thesis. This chapter ends with explaining the organization of the rest of the thesis.

Chapter 2 discusses the background and preliminaries. We start the chapter defining basic graph terms such as a graph, clique of a graph, bipartite graph. Then we illustrate about sparse matrix, Jacobian matrix and direct determination. Next, we explain how bipartite graph coloring relates to matrix problems by discussing algorithmic differentiation. After that, graph coloring is defined. Then the representation of the partition problem as a graph coloring problem and the comparison between uni-directional and bi-directional partitioning are illustrated. This chapter ends with discussing complete direct cover based on which we developed our coloring algorithm.

Approaches to the partition algorithm are illustrated in Chapter 3. In the beginning, different data structures such as Compressed Sparse Row (CSR), Compressed Sparse Column (CSC), and Bucket data structure used in DSJM are described. Then different ordering and partitioning algorithms are introduced, which we used in the Star bi-coloring algorithm. Smallest last Ordering (SLO), Largest First Ordering (LFO), and Incidence Degree Ordering are discussed using an example.

Chapter 4 describes the implementation of this thesis. Different steps of the implementation, such as degree calculation of bipartite graph, distance-2 neighbor calculation, the formation of the groups, updating degree are illustrated in this chapter. Then the heuristic approach for star bi-coloring is introduced. The chapter ends with the verification of the coloring.

Numerical experiments are discussed in Chapter 5. The efficiency of our data structure and algorithm are demonstrated using the numerical results. We compared our results with several previous research works on coloring algorithms.

Chapter 6 wraps up the thesis with the concluding summary of the research work. Some future research directions are also listed in this chapter.

Chapter 2

Background and Preliminaries

In this chapter, we will discuss some definitions and background works which will be beneficial to understand the thesis.

2.1 Graph

A graph is generally known as a network. Graphs can represent all real-world systems. For example, a city's transit route can be represented as a graph where bus stops can be denoted by vertices and connecting roads between two bus stops can be represented by edges.

A graph is represented by G (V, E) where V is the set of vertices and E is the set of edges. Suppose there are two vertices u,v ε V. If u,v is connected by an edge such that (u,v) ε E, then they are adjacent to one another.

In figure 2.1, a graph is represented with 6 vertices V = $\{u_1, u_2, u_3, v_1, v_2, v_3\}$ and 5 edges E = $\{\{u_1, v_1\}, \{u_1, v_3\}, \{u_2, v_2\}, \{u_3, v_2\}, \{u_3, v_3\}\}$. In the figure, vertices are represented as circle and edges are represented as lines connecting two vertices.

Adjacent vertices are connected using an edge. Adjacent nodes are neighbors of each other. In the figure 2.1, u_1 and v_1 adjacent to each other. The number of neighbors of a vertex is known as its degree. Degree of u_1 is 2, degree of u_2 is 1, degree of v_3 is 2.

In this thesis, we worked with simple graphs. Simple graphs do not contain multiple edges, self-loop.

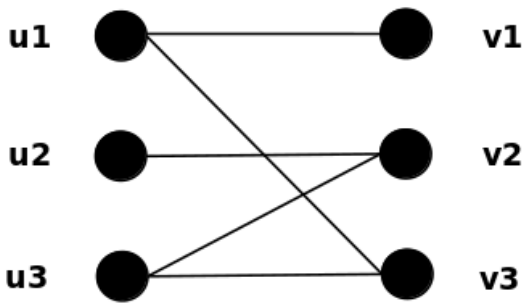

Figure 2.1: Graph

2.2 Clique of a Graph

A clique of a graph is a complete sub-graph of that graph. In a complete graph, all the vertices are connected to each other. For a graph G(V, E), clique is the complete sub-graph $G_s(V_s, E_s)$ where V_s are the member of V and E_s are the members of E, and all the vertices of V_s are connected to each other.

There can be multiple cliques in a graph. The clique having the maximum number of vertices is known as a maximum clique.

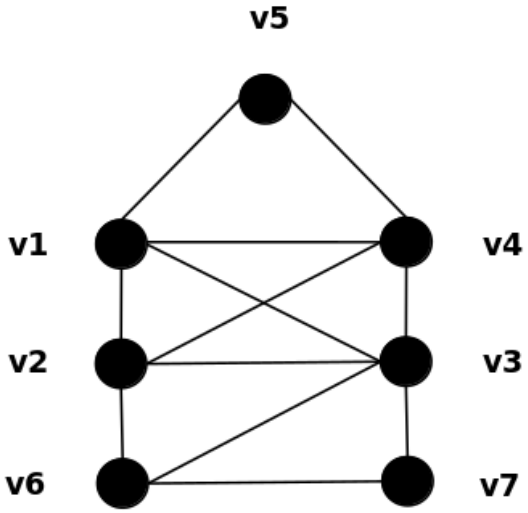

Figure 2.2: Clique of a graph

In figure 2.2, there are several cliques such as $\{v_1, v_4, v_5\}$, $\{v_1, v_2, v_3, v_4\}$, $\{v_1, v_2, v_4\}$, $\{v_1, v_2, v_3\}$, $\{v_1, v_4, v_5\}$, $\{v_1, v_2, v_4\}$, $\{v_2, v_3, v_6\}$. Maximum clique is $\{v_1, v_2, v_3, v_4\}$ and

maximum clique size is four. Maximum cliques are sometime referred as cliques [14].

2.3 Bipartite Graph

Bipartite graph are formed using two disjoint sets of vertices. Suppose a bipartite graph is G (V_c, V_r, E). V_c and V_r are two disjoint sets of vertices and there are edges between $V \subset V_c$ to $V \subset V_r$. There is no edge between the vertices of V_c or between the vertices of V_r.

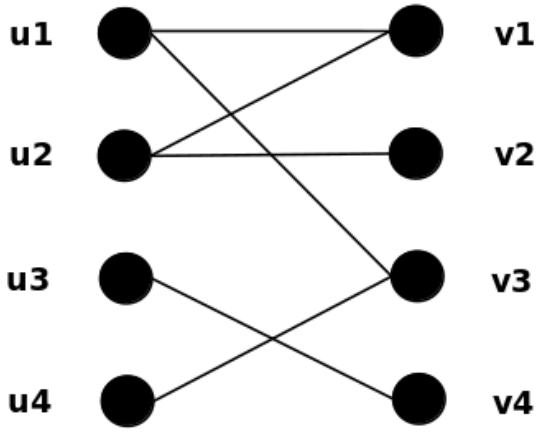

Figure 2.3: Bipartite graph

Figure 2.3 represents a bipartite graph. The bipartite graph in figure 2.3 has two disjoint sets of vertices U = $\{u_1, u_2, u_3, u_4\}$ and V = $\{v_1, v_2, v_3, v_4\}$. The edges of this graph are E = $\{\{u_1, v_1\}, \{u_1, v_3\}, \{u_2, v_1\}, \{u_2, v_2\}, \{u_3, v_4\}, \{u_4, v_4\}\}$. There are edges between the vertices of U and the vertices of V.

The graph in figure 2.4 does not look like a bipartite graph at first. But the vertices of the graph in figure 2.4 can be divided into two disjoint sets U and V where edges are between U and V.

The bipartite graph in figure 2.4 has two disjoint sets of vertices U = $\{u_1, u_2, u_3, u_4, u_5\}$ and V = $\{v_1, v_2, v_3, v_4, v_5\}$.

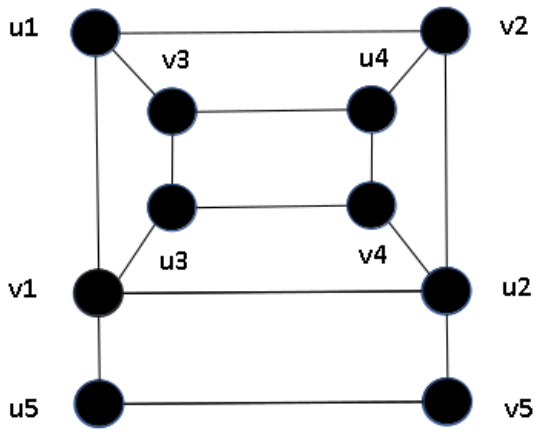

Figure 2.4: Complex Bipartite graph

2.4 Sparse Matrix

A matrix containing few non-zero values is known as a sparse matrix. In a sparse matrix, there are more zero-valued elements than non-zero valued elements.

The following matrix is a sparse matrix.

$$A = \begin{pmatrix} 5 & 0 & 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 7 & 0 \\ 0 & 0 & 4 & 0 & 0 & 0 \\ 0 & 0 & 0 & 5 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 9 \end{pmatrix} \tag{2.1}$$

Computational advantage can be ensured from the knowledge of many zero entries. We may store only non-zero values of the matrix to make the computation faster.

Sparse matrix bcspwr03 collected from [2] is shown in figure 2.5. Suppose there is an m by n matrix. The (1,1) element is represented in the top left corner, and (m,n) element is shown in the bottom right corner in the rectangular structural plot. The black rectangles represent non-zero entries. bcspwr03 has 118 rows, 118 columns, and 297 non-zero entries.

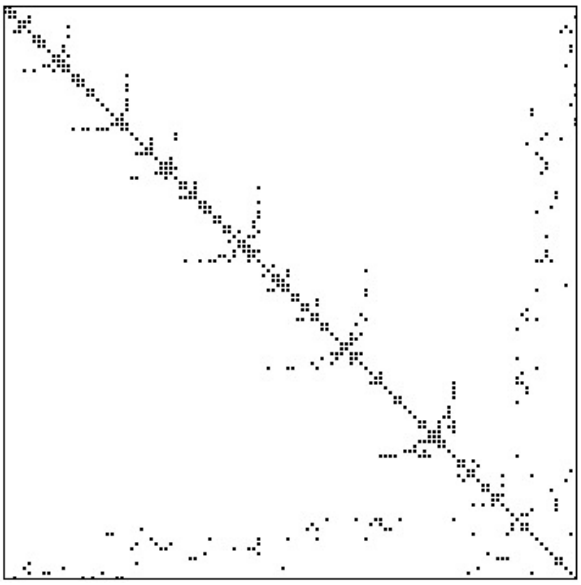

Figure 2.5: Structure plot of bcspwr03 (sparse matrix)

2.5 Jacobian Matrix

The first order partial derivative of a vector valued function is the jacobian matrix of that vector valued function. Suppose $F : \mathbb{R}^n \to \mathbb{R}^m$ is a continuously differentiable vector valued function. Vector $x \in \mathbb{R}^n$ is the input of the function and vector $F(x) \in \mathbb{R}^m$ is the output of the function. Jacobian J of F where $F = (f_1, f_2, f_3, ...f_m)^T$ is-

$$J = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & .... & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & .... & \frac{\partial f_2}{\partial x_n} \\ \frac{\partial f_3}{\partial x_1} & \frac{\partial f_3}{\partial x_2} & .... & \frac{\partial f_3}{\partial x_n} \\ .... & .... & .... & .... \\ \frac{\partial f_m}{\partial x_1} & \frac{\partial f_m}{\partial x_2} & .... & \frac{\partial f_m}{\partial x_n} \end{pmatrix} \tag{2.2}$$

2.6 Direct Determination

The evaluation of mathematical derivatives is often required to solve the problems in nonlinear optimization and differential equations. We are approaching the problem, determination of Jacobian matrix $F'(x)$ of a once continuously differentiable mapping F :

$\mathbb{R}^n \to \mathbb{R}^m$ at a given point $x\varepsilon\mathbb{R}^n$. The product of the Jacobian matrix with a vector s may be estimated as

$$\left.\frac{\partial F(x+ts)}{\partial t}\right|_{t=0} = F'(x)s \equiv As \approx \frac{1}{\varepsilon}[F(x+\varepsilon s) - F(x)] \equiv b, \tag{2.3}$$

with one additional calculation of F at $F(x+\varepsilon s)$, where $\varepsilon > 0$ is a small addition. We are assuming that F(x) is already being computed. Forward mode of algorithmic differentiation [18] gives the value b = $F'(x)S$. The numerical value b is correct up to the machine round off. The cost of calculating b is a small multiple of the cost of one function evaluation. In conclusion we can say that, the product of the Jacobian matrix of function F at a point x with a vector s can be evaluated as AS. The cost of calculating AS is a small multiple of the cost of evaluating the function F(x). Therefore the computational cost of determining a Jacobian matrix can be represented in terms of the number of matrix-vector products in the form of AS.

$$A = \begin{pmatrix} a_{11} & 0 & a_{13} & 0 \\ a_{21} & a_{22} & 0 & 0 \\ 0 & 0 & 0 & a_{34} \\ a_{41} & a_{42} & 0 & 0 \end{pmatrix} \tag{2.4}$$

$$S = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 1 \\ 1 & 0 \end{pmatrix} \tag{2.5}$$

Consider the sparse matrix A in equation (2.4). We are defining matrix S where S(:,1) = $e_1 + e_4$, and S(:,2) = $e_2 + e_3$ reproduced from [9] where e_i denotes the i-th coordinate vector. Here the colon notation [12] is used to represent submatrices.

$$B = AS = \begin{pmatrix} a_{11} & 0 & a_{13} & 0 \\ a_{21} & a_{22} & 0 & 0 \\ 0 & 0 & 0 & a_{34} \\ a_{41} & a_{42} & 0 & 0 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} a_{11} & a_{13} \\ a_{21} & a_{22} \\ a_{34} & 0 \\ a_{41} & a_{42} \end{pmatrix} \tag{2.6}$$

Now compute the product B = AS using the finite difference formula from equation (2.3). The multiplication is shown in equation (2.6). Forward difference (FD) estimation of the unknown elements of A can be read-off from the compressed matrix B at a cost of only 2 (instead of 4) additional evaluations of function F. Matrix A is said to be directly determined if all the non-zero elements can be read-off from compressed matrix B.

2.7 Algorithmic Differentiation

Algorithmic Differentiation (AD) is the approach to differentiate a function F to compute derivative matrices, where F is given by a computer program. The cost of determining the Jacobian matrix depends on the cost of algorithmic differentiation. Determining the sparse Jacobian matrix would be faster by using the sparsity pattern. Algorithmic differentiation, which is also known as automatic differentiation determines derivatives of a function based on arguments with no truncation error. Algorithmic differentiation is a chain based rule method.

Let f be a function of the vector $y \in \mathbb{R}^m$, which is a function of the vector $x \in \mathbb{R}^n$. The derivative of f with respect to x using chain rule is-

$$\nabla_x f(y(x)) = \sum_{i=1}^{m} \frac{\partial f}{\partial y_i} \nabla y_i(x) \tag{2.7}$$

where ∇ denotes the gradient. At a time one or two arguments are used in sequence of operations during the evaluation of automatic differentiation. We demonstrate the basic

algorithm by the following example borrowed from [9].

$$f(x_1,x_2) = x_1 sin(x_2) + x_2 cos(x_1) \tag{2.8}$$

A sequence of arithmetic operations are evaluated to compute f-

$$\begin{aligned} v_1 &= x_1 \\ v_2 &= x_2 \\ v_3 &= sin(x_2) \\ v_4 &= cos(x_1) \\ v_5 &= v_1 * v_3 \\ v_6 &= v_2 * v_4 \\ v_7 &= v_5 + v_6 \end{aligned} \tag{2.9}$$

where $v_i, i = 3,4,5,6$ are intermediate quantities. The result of computation is obtained in-

$$f(x_1,x_2) = v_7 \tag{2.10}$$

The sequence of operations in equation (2.9) are known as code lists. It is possible to form different code lists from same function. The rules of differentiation can be applied to evaluate the derivative of a function with respect to the variables x_1 and x_2.

Automatic differentiation has two modes: forward mode and backward mode. Forward mode of automatic differentiation is able to determine the non-zero values in a column group, and reverse mode can determine the non-zero entries in row groups. The cost of determining the Jacobian matrix depends on the cost of algorithmic differentiation.

2.8 Column Intersection Graph

A matrix can be represented as a column intersection graph. Every column of the matrix is considered as a vertex in the graph. There will be an edge between two vertices (columns) when there are non-zero entries in both columns in the same row position.

Figure 2.6 represents the column intersection graph of the following matrix B.

$$B = \begin{pmatrix} 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 \end{pmatrix} \tag{2.11}$$

There are five columns in the matrix. For that the graph in figure 2.6 has five vertices V=$\{v_1, v_2, v_3, v_4, v_5\}$. Column 1 and column 4 of matrix B has non-zero entries in the same row. For that v_1 and v_4 has an edge in the graph. Also column 1 has overlapping non-zero entry with column 3. So there is an edge between v_1 and v_3. In this manner, the column intersection graph is constructed.

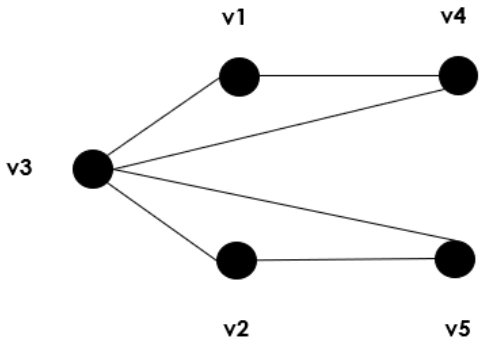

Figure 2.6: Column Intersection Graph

2.9 Graph Coloring

Vertices of a graph is colored in a way that no two neighboring vertices are appointed the same color. A graph G (V,E) is p-colorable when there is a function $\phi : V \rightarrow \{1, 2, ..., p\}$

such that $\phi(u) \neq \phi(v)$, when $\{u, v\} \in E$.Optimal coloring is achievable using minimum number of colors. The minimum number of colors required to color a graph is known as chromatic number X(G) of a graph. Whether a graph is p-colorable or not, is a NP-Complete prblem [8]. Suppose, G(U ∪ V, E) is a bipartite graph. Here the vertices are split into two

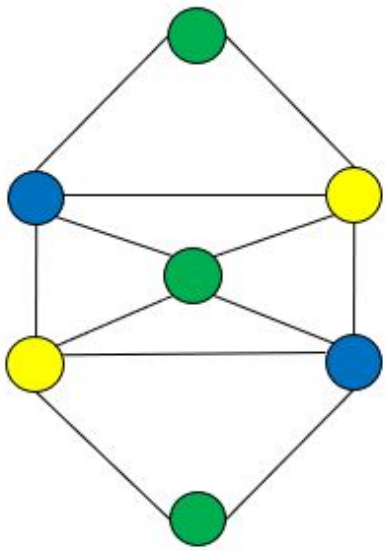

Figure 2.7: Graph colored using 3 colors

disjoint sets U and V. G is path p-colorable, if there are atleast three colors required to color a path of length three. Also

$$\{\phi(u) : u \in U\} \cap \{\phi(v) : v \in V\} = \varnothing \tag{2.12}$$

2.10 Representation of Partitioning Problem as a Graph Coloring Problem

A sparse matrix $A \in \mathbb{R}^{m \times n}$ has m rows and n columns. This matrix can be represented as a bipartite graph G(U ∪ V, E) where columns of the matrix are denoted as the vertices in U = $\{c_1, c_2, c_3, ..., c_n\}$ and rows of the matrix are denoted as the vertices in V = $\{r_1, r_2, r_3, ..., r_n\}$. Total number of vertices in the bipartite graph will be the summation of

number of rows and number of columns of the matrix. There is an edge between r_i and c_j, when there is a non-zero value at $a_{i,j}$ of matrix A.

$$A = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix} \tag{2.13}$$

Bipartite graph representation of a sparse matrix A is shown in figure 2.8.

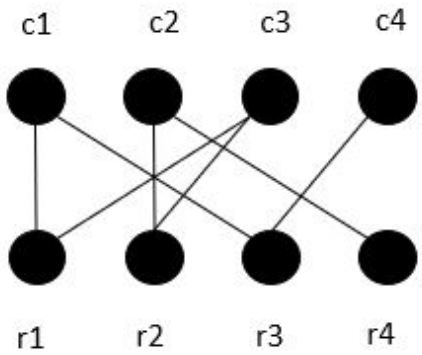

Figure 2.8: Bipartite graph representation of matrix A

Matrix A has four rows and four columns. Bipartite graph in the figure 2.8 has eight vertices. The vertices are split into two disjoint sets U = $\{c_1, c_2, c_3, c_4\}$ and V = $\{r_1, r_2, r_3, r_4\}$. Total number of edges in the bipartite graph is seven which is the total number of non-zeroes in the matrix A.

Theorem 2.1 on path p-coloring of a bipartite graph based on a matrix is discussed in [17].

Theorem 2.1. *Suppose 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 $G_b(A)$.*

2.11 Uni-directional partitioning vs Bi-directional partitioning

In uni-directional partitioning, sparsity can be exploited either in columns or rows. The sparsity pattern of a sparse matrix can be exploited in both rows and columns using bi-directional partitioning. The following example shows that using bi-directional partitioning to determine non-zero elements in a sparse matrix works more efficiently than uni-directional partitioning.

$$A = \begin{pmatrix} 0 & 0 & a_{13} \\ 0 & 0 & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \tag{2.14}$$

In the beginning, we are considering uni-directional partitioning based on columns. Three column groups consisting of each column in each group are needed to determine all the non-zeroes of matrix A of equation (2.7). Three groups are required as a_{31}, a_{32} and a_{33} are in same row. For that column 1, column 2 and column 3 can not be grouped together.

In the same manner, three groups are needed to determine all the non-zeroes of A using uni-directional partitioning based on rows.

All the non-zeroes can be determined using two groups in bi-directional partitioning. Group 1 consisting column 3 determines a_{13}, a_{23} and a_{33}. Group 2 having row 3, row 1 and row 2 determines a_{31} and a_{32}. Therefore, bi-directional partitioning require less group than uni-directional partitioning to determine all the non-zero elements of a sparse matrix.

2.12 Complete Direct Cover

Hossain and Steihaug [17] introduced the direct cover property regarding the approximation of the Jacobian matrix. Following [17], we are reproducing the definition of complete direct cover.

We assume that there is an m by n sparse Jacobian matrix (A) where non-zero elements are denoted as a_{ij}. Suppose, there is a group Z of rows or columns in a row-column

partition.

Definition 2.2. A non-zero a_{ij} is covered by Z where row $r_i \in Z$ or column $c_j \in Z$.

Definition 2.3. A non-zero a_{ij} covered by Z is said to be directly determined by Z when there are no column in Z (except c_j) that has a non-zero in row r_i.

Definition 2.4. Suppose, V_c is the collection of subsets of columns and V_r is the collection of subsets of rows. Then there will be complete direct cover set [V_c,V_r] of sparse matrix A if-

- The intersection of any two subsets is empty.
- For each nonzero element $a_{i,j}$ of a sparse matrix A, there is a subset $Z \in V_c \cup V_r$, such that $a_{i,j}$ is directly determined by Z.

Chapter 3

Approaches to Partitioning Algorithm

In this chapter, we discuss the different partitioning algorithms defined in DSJM [16]. We will explain the largest first ordering, smallest first ordering, and incidence degree ordering. All of these partitioning and ordering algorithms scan the columns based on degree and colors sequentially using minimum colors.

At the beginning of the chapter, the data structures used in the implementation are discussed. The data structures, such as Compressed Sparse Row, Compressed Sparse Column, and Bucket Data Structure, are defined in DSJM [16]. These data structures are also demonstrated using detailed examples.

Then the ordering and partitioning algorithms are illustrated using graphical examples.

3.1 Data Structures

Here we are going to discuss the data structures used for the implementation. The sparse matrix contains a few numbers of non-zero entries. Storing the entire sparse matrix is not cost-effective as it contains a lot of zero-valued elements. We will discuss the data structures used to store the sparse matrix efficiently. Also, we will discuss the bucket data structure. The bucket data structure is used for the implementation of different partitioning and ordering algorithms.

3.1.1 Compressed Sparse Row (CSR)

The compressed sparse row [16] is a data structure that stores the non-zero entries of a sparse matrix efficiently.

Compressed sparse row (CSR) is implemented using three arrays :

- value: value array stores the values of matrix elements.
- rowptr: rowptr points the column indices.
- colind: colind stores the column index of non-zero elements of the matrix.

For a sparse matrix $\mathbb{R}^{m\times n}$, where the number of rows is m, and the number of columns is n, the size of rowptr is m+1. Suppose there is nnz number of non-zero entries in the sparse matrix. The size of the value array and the size of the colind is nnz. In CSR, rowptr and colind are integer arrays. Let a_{ij} is a non-zero entry in the matrix A. To access the value, rowptr(i) is retrieved first. rowptr(i) returns the starting column index for ith row. The number of non-zero entries in the i-th row can be computed by-

$$\text{number of non-zero elements in i-th row} = rowptr(i+1) - rowptr(i) \tag{3.1}$$

The elements of the i-th row can be accessed using-

$$\text{colind(rowptr(i)) to colind(rowptr(i+1) - 1)} \tag{3.2}$$

Suppose there is a matrix A.

$$A = \begin{pmatrix} a_{11} & 0 & a_{13} & 0 & 0 & 0 & 0 & 0 \\ 0 & a_{22} & 0 & 0 & a_{25} & 0 & a_{27} & 0 \\ 0 & 0 & 0 & 0 & a_{35} & 0 & 0 & 0 \\ 0 & 0 & a_{43} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & a_{54} & 0 & 0 & 0 & 0 \\ a_{61} & 0 & 0 & 0 & 0 & a_{66} & 0 & 0 \\ 0 & 0 & 0 & a_{74} & 0 & 0 & 0 & 0 \\ 0 & a_{82} & 0 & 0 & 0 & 0 & a_{87} & a_{88} \end{pmatrix} \tag{3.3}$$

Figure 3.1 is showing the bipartite graph representation of the above matrix. There are eight rows, and eight columns in the matrix A. Rows and columns formed two disjoint sets to construct the bipartite graph. There are eight vertices in the upper portion in figure 3.1, which represents columns, and there are eight vertices in the lower portion, which represents the rows. a_{11} is a non-zero element in the matrix. There is an edge between row 1 and column 1. Also, there is an edge between row 2 and column 5, as a_{25} is a non-zero. There are fourteen edges in the bipartite graph as there are fourteen non-zeroes in matrix A.

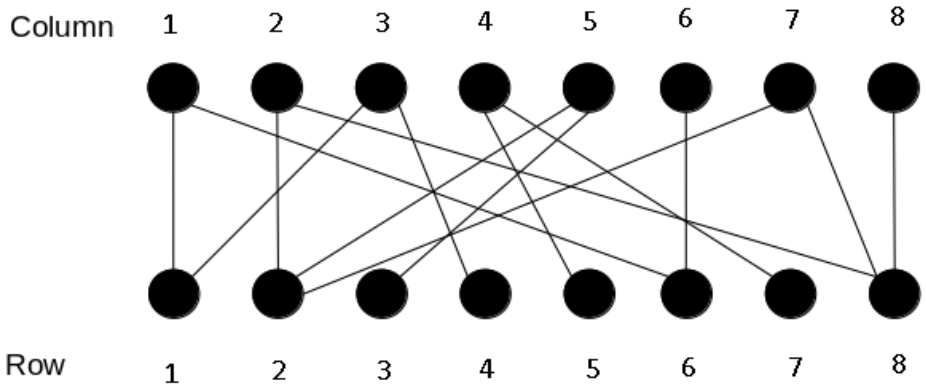

Figure 3.1: Bipartite graph representation of matrix A

Matrix A is stored using a compressed sparse row, and it is graphically displayed in figure 3.2. The size of the rowptr is nine, which is m+1 (m=8). The value array is storing 14 non-zero values. colind has fourteen column indices where the non-zero exists. Suppose we would like to determine the number of non-zero entries for row 2. There is rowptr(3) - rowptr(2) = 6 -3 = 3 non-zero entries in row 2. 2nnz + m + 1 memory locations are needed

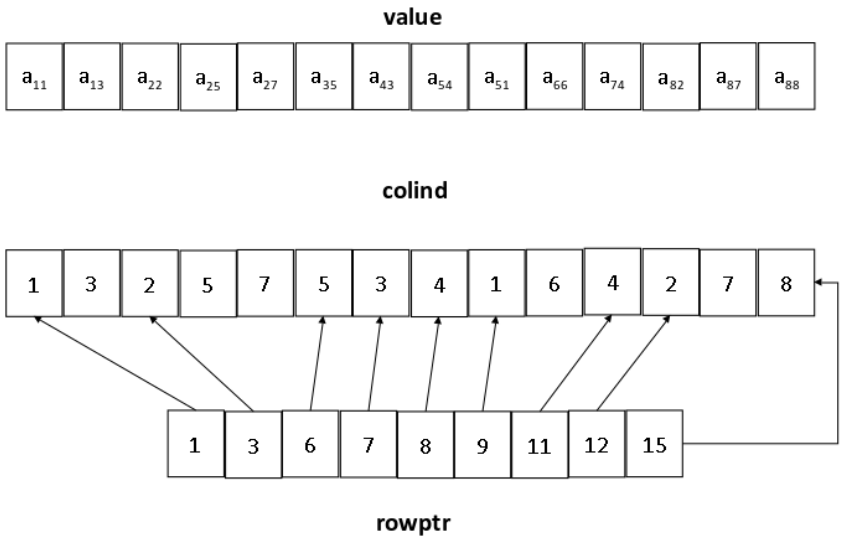

Figure 3.2: Compressed sparse row (CSR) representation of matrix A

to store the non-zeroes of a matrix using compressed sparse row.

3.1.2 Compressed Sparse Column (CSC)

Compressed sparse column (CSC) [15] is column based data structure to store the non-zeroes of sparse matrix. CSC is also implemented using three arrays-

- value: value array stores the values of matrix elements.
- colptr: colptr points the row indices.
- rowind: rowind stores the row index of non-zero elements of the matrix.

In CSC, colptr and rowind are integer arrays as they point to an array index. Figure 3.3 shows the CSC representation of matrix A from equation (3.3). There are eight columns

in the matrix. The size of the colptr is nine, which is n+1 (n=8). The size of rowind is fourteen, as there are fourteen non-zeroes in matrix A.

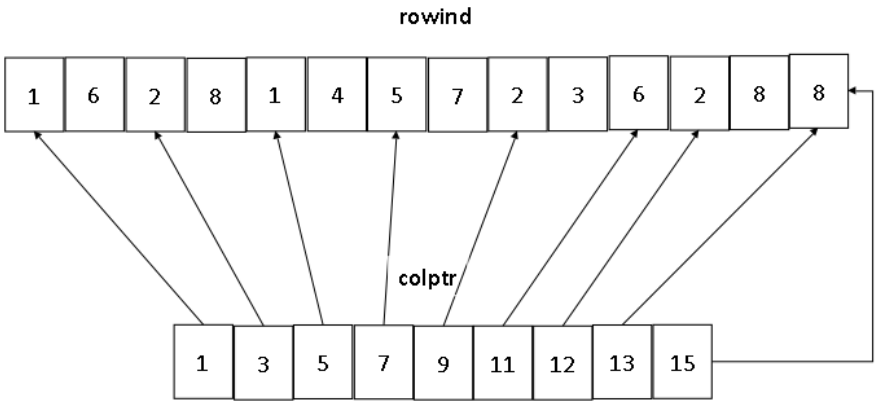

Figure 3.3: Compressed sparse column (CSC) representation of matrix A

The number of non-zero entries in the j-th column can be computed by-

$$\text{number of non-zero elements in j-th column} = colptr(j+1) - colptr(j) \tag{3.4}$$

The elements of the j-th column can be accessed using-

$$\text{rowind(colptr(j)) to rowind(colptr(j+1) - 1)} \tag{3.5}$$

Suppose we would like to find out the number of non-zeroes in column 3. There are colptr(4) - colptr(3) = 7 -5 = 2 non-zero entries in column 3.

nnz + n +1 memory locations are needed to store the sparse matrix using CSC when the number of columns is n, and the number of non-zeroes is nnz.

Harwell-Boeing collection at the matrix market [6] is represented using CSC. Matlab also uses CSC to represent sparse matrices [11].

In our implementation, both CSR and CSC have been used. Therefore, 3nnz + m + n + 2 memory locations are needed to store the sparse matrix.

3.1.3 Bucket Data Structure

Apart from CSR and CSC, bucket [22] is the frequently used data structure in the partitioning and ordering algorithms. The bucket data structure is also used in the coloring algorithm, along with CSR and CSC. The bucket data structure is implemented using three arrays and multiple degree lists. Based on the degrees of vertices, a multiple degree list is constructed. Three arrays of the bucket data structure are-

- HEAD: HEAD array stores the first vertex of each degree list.
- PREVIOUS: PREVIOUS array stores the previous vertex of the processing vertex.
- NEXT: NEXT array stores the next vertex of the current vertex.

The size of the HEAD is the maximum degree (maxdeg) of the vertex in a graph. The size of the NEXT and PREVIOUS is n+1, where n is the number of columns. So, in summation, maxdeg + 2n + 2 memory locations are required to use the bucket data structure.

Figure 3.4 represents the column intersection graph of the following matrix A.

$$A = \begin{pmatrix} a_{11} & 0 & 0 & a_{14} & 0 \\ 0 & a_{22} & a_{23} & 0 & a_{25} \\ a_{31} & 0 & a_{33} & a_{34} & 0 \\ 0 & a_{42} & 0 & 0 & a_{45} \end{pmatrix} \tag{3.6}$$

Figure 3.5 represents the bucket representation of matrix A. In figure 3.5, ndeg is an array that stores the degree of each vertex. This degree information is required at the beginning of some partition algorithms, such as the smallest last ordering (SLO). After the first iteration, this degree information is not required in SLO. Degree information can be retrieved from degree lists. However, some partitioning algorithms, such as incidence degree ordering (IDO), require the degree information during the whole procedure. ndeg (1) = 2 means that the degree of vertex 1 is 2. There are two-degree lists on this example. Vertices 1,2,4 and 5 have degree 2. Vertex 3 has degree 4.

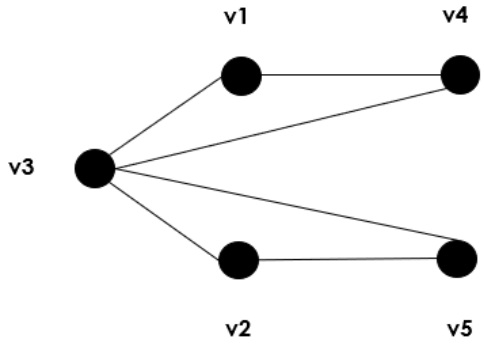

Figure 3.4: Graph representation of matrix A

HEAD(2) = 5 points that, vertex 5 is the first vertex in degree list 2. Also, HEAD(4) = 3 represents that, vertex 3 is the first vertex in the degree list 4.

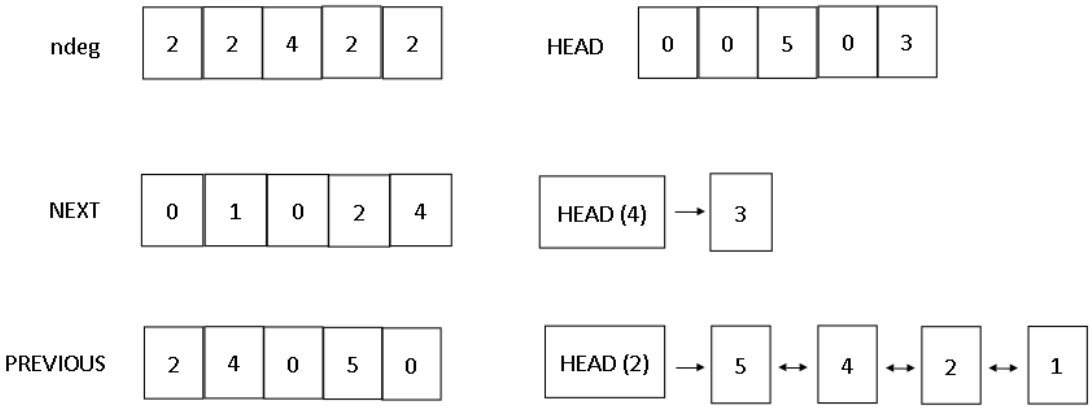

Figure 3.5: Bucket representation of matrix A

Degree lists can be traversed using PREVIOUS and NEXT array. NEXT(2) = 1 means that, the next vertex of vertex 2 is vertex 1. Also, NEXT(3) = 0 means that there is no vertex after vertex 3. Suppose PREVIOUS(1) = 2. It denotes that the previous vertex of vertex 1 is vertex 2. When PREVIOUS(3) = 0, it represents that there is no vertex before vertex 3.

3.2 Partition Algorithms

Traversing vertices in a specific order during coloring of vertices may lead to better outcomes [9]. A better outcome means a fewer number of groups may be needed to color

or group all the vertices of a graph. Apart from the natural order, we have used SLO, LFO, and IDO in order to scan the vertices in our proposed coloring algorithm. Partition and ordering algorithms are implemented using a compressed sparse row (CSR), compressed sparse column (CSC), and bucket data structure. Matrix A from equation (3.6) and the graph representation of matrix A from figure 3.4 are used to discuss the smallest last ordering (SLO), largest first ordering (LFO), and incidence degree ordering (IDO).

3.2.1 Smallest Last Ordering (SLO)

Suppose V' is the set of ordered vertices where $V' = \{v_n, v_{n-1}, ..., v_{i+1}\}$. The vertex v_i is placed in the ordered set of vertices, which was an unordered vertex u such that deg(u) is minimum in $G[V \backslash V']$. The unordered set of vertices is in $G[V \backslash V']$. The algorithm [22] of the smallest last ordering (SLO) is shown in figure 3.6.

```
SLO (S(A), order)
1       slindex ← n
2       j ← {1,2,....,n}
3       while j ≠ ∅
4           let i ∈ j be such that d_j(l) is minimum
5           order(slindex) ← l
6           slindex ← slindex – 1
7           j ← j \ {l}
```

Figure 3.6: SLO Algorithm

The input of SLO is the sparsity pattern *S(A)* of the matrix A and the output is the *order* array, which stores the ordering of vertices returned by SLO.

In SLO, vertex with the smallest degree is stored in the last position of order array. For that, slindex is initialized to n in the algorithm in figure 3.6. j is an array consisting of all the column vertices. Loop in line 3 executes until all the column vertices are traversed. SLO processes the vertex with a minimum degree at first and assigns that vertex to the last position of order array. Then slindex gets updated by decremented by 1. Line 7 removes

the ordered vertex from the unordered set of vertices. While loop in line 3 continues until all the vertices assigned in the order array. This algorithm is for column vertices. We have adopted this algorithm for row vertices.

Now we will discuss SLO using the matrix A from equation (3.6) and the graph representation of matrix A from figure 3.4.

Figure 3.7 represents the data structures after the initialization step of SLO. ndeg array has the degree information of every vertex. Initially, there are two-degree lists, which are HEAD (2) and HEAD(4). HEAD array contains the first vertex of each degree list. HEAD (2) = 5 means that, degree list (2) starts with vertex 5. In the beginning, all the vertices are unordered. So V' is initially empty. Degree lists can be traversed using the NEXT and PREVIOUS array.

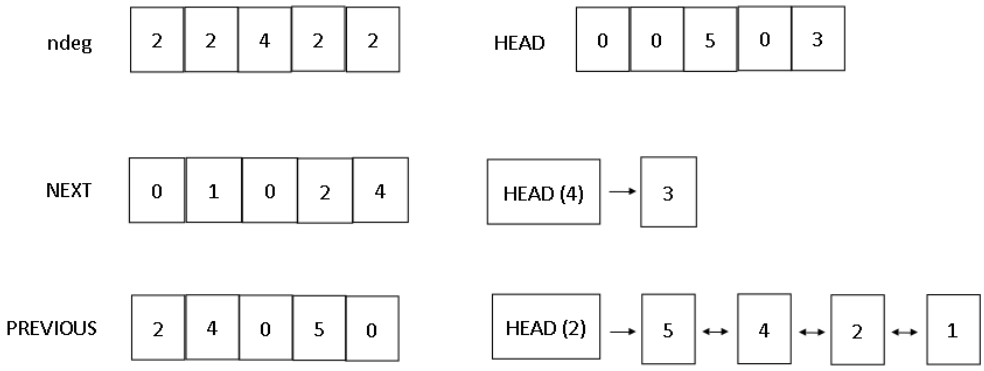

Figure 3.7: Data Structures after initialization of matrix A (SLO)

SLO starts processing with the vertex having the smallest degree. Among the two-degree list 2 and 4, 2 is the smaller one. In degree list (2), there are 4 vertices. Nevertheless, HEAD(2) is 5. Therefore, 5 is placed on the last position of the order array. There are edges $\{5,2\}$ and $\{5,3\}$. So the degrees of vertex 2 and 3 will be decreased. 2 will move to the degree list (1), and 3 will move to the degree list (3). HEAD, NEXT, and PREVIOUS are also updated as degree lists are changed. The updated data structures after the first iteration are shown in figure 3.8.

In the second iteration vertex, 2 will be added to the order array as it has the lowest

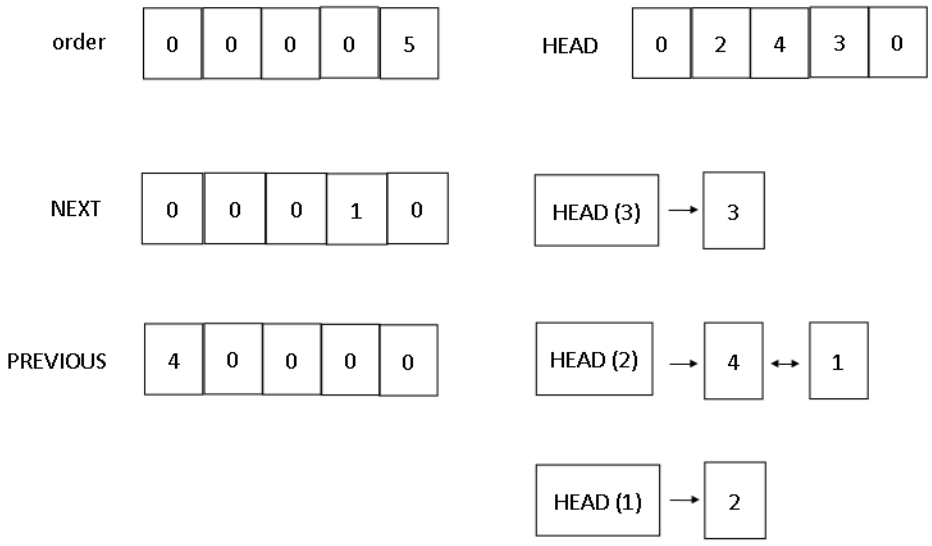

Figure 3.8: Data Structures after first iteration (SLO)

degree. There are edges $\{2,3\}$. The degree of vertex 3 will be reduced and move to the degree list (2). The updated data structures are shown in figure 3.9.

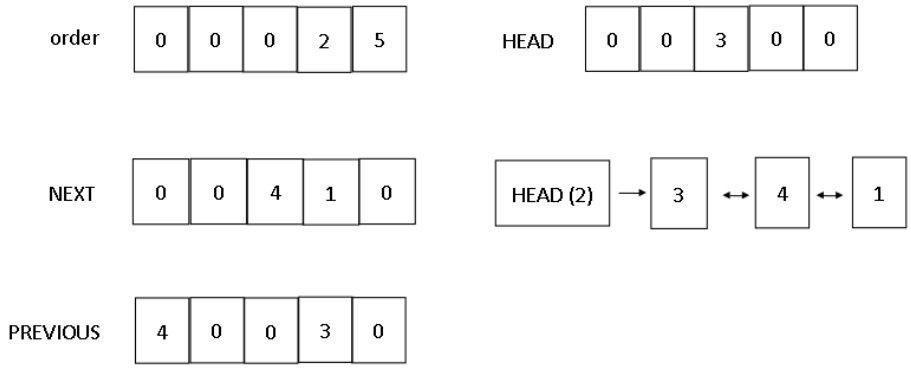

Figure 3.9: Data Structures after second iteration (SLO)

At this moment lowest degree is 2 and HEAD(2) = 3. Vertex 3 is removed from the unordered set and is added to the order array. The degree of vertex 1 and 4 are updated as they have edges with vertex 3. The new values of data structures looks like figure 3.10.

Vertex 1 will be processed next as HEAD(1) = 1. 1 will be added to the order array, and the degree of 4 will be updated. The updated data structures are represented in figure 3.11.

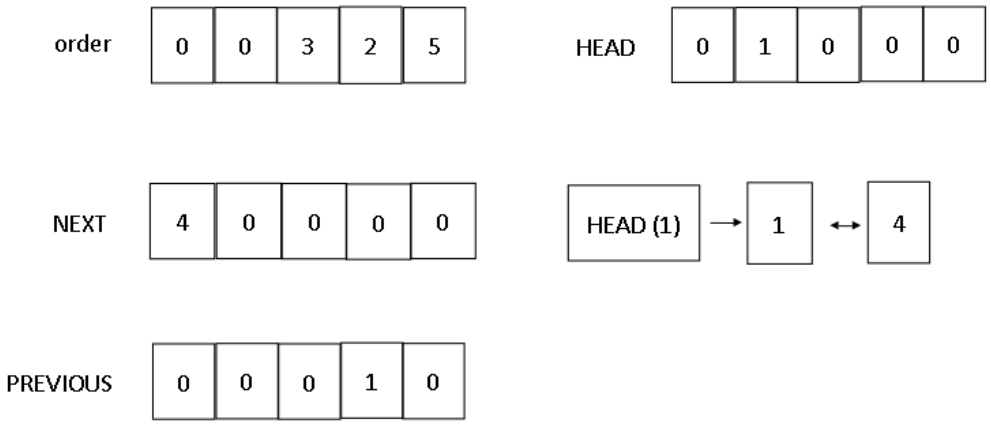

Figure 3.10: Data Structures after third iteration (SLO)

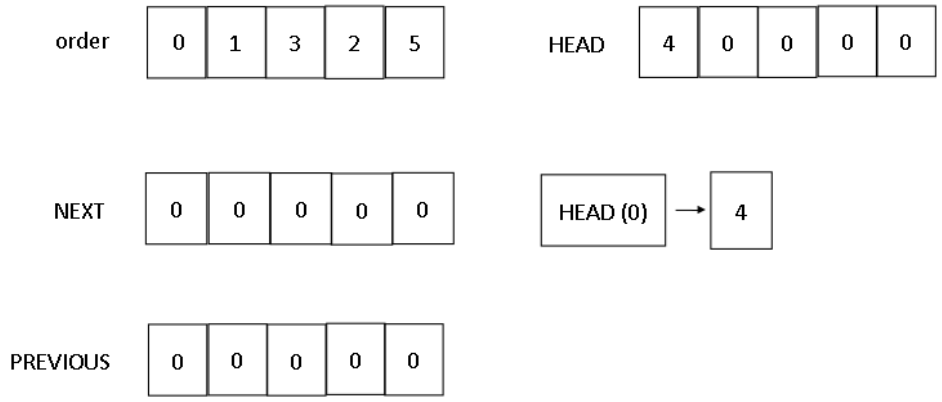

Figure 3.11: Data Structures after fourth iteration (SLO)

Only vertex 4 is remaining in the unordered list. Vertex 4 is added to the order array, and the final ordering is shown in figure 3.12.

Figure 3.12: Final ordering of vertices (SLO)

3.2.2 Incidence Degree Ordering (IDO)

Let us assume that $V' = (v_1, v_2, ..., v_{i-1})$ is an ordered set of vertices. The algorithm adds the next vertex u in the i-th position of ordered set from the unordered set of vertices where deg(u) is maximum in G[V']. There might be more than one vertex with maximum degree. Vertex with a maximum degree in G[$V \backslash V'$] is chosen to break the tie.

The algorithm of Incidence Degree Ordering is listed in figure 3.13.

```
IDO (S(A), order)
1     idindex ← 1
2     j ← {1,2,...,n}
3     u ← j
4     O ←  ∅
5     while u ≠ ∅
6         M  ← {l ∈ u | d_O(l) is maximum}
7         let j ∈ M be such that du(j) + d_O(j) is maximum
8         order (idindex) ← j
9         idindex ← idindex + 1
10        O ← O ∪ {j}
11        u ← u\{j}
```

Figure 3.13: IDO Algorithm

Sparsity pattern *S(A)* of the matrix A is the input of IDO, and the ordering of vertices returned by IDO is stored in *order* array is the output.

The first ordered vertex of IDO is stored in the first position of order array. For that, idindex starts at 1 in line 1 of the algorithm in figure 3.13. In IDO, two-degree values are considered. Incidence degree is calculated from induced subgraph, and the vertices in induced subgraph are O in line 4. An unordered set of vertices are u in line 3. While loop in line 5, iterates until all the vertices are added in the order array. Vertex has the maximum sum of incidence degree, and normal degree is assigned in the first position of order array

using line 6. Line 10 updates the incidence degree of vertices, and line 11 updates the unordered set of vertices.

Figure 3.14 shows the initialized Data Structures of matrix A for IDO. In the beginning, there is no vertex in the ordered set V'. Incidence degree is the degree of unordered vertices in the subgraph induced by ordered vertices of graph G[V']. The incidence degree of all the vertices is 0. For that, there is an only one-degree list (0). To break the tie, we need the degree of unordered vertices in G[$V \backslash V'$]. These degrees are stored in ideg.

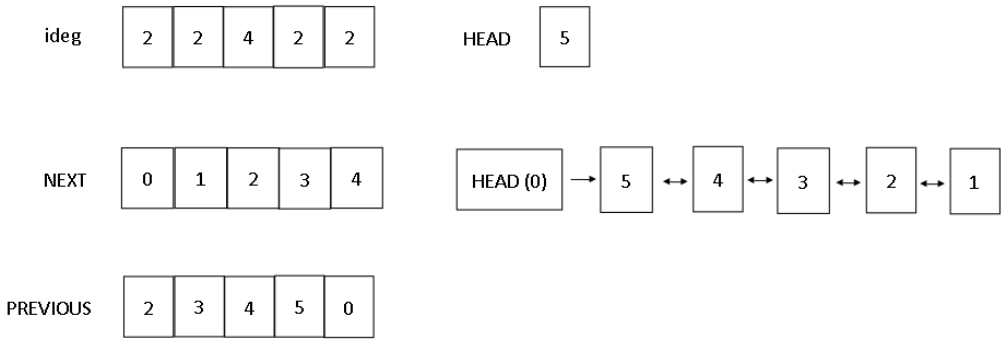

Figure 3.14: Data Structures after initialization of matrix A (IDO)

In the first iteration, the algorithm searches for a vertex with the maximum incidence degree. However, at this moment, all the vertices have the same incidence degree. ideg is used to break the tie. From ideg, we can see that vertex 3 has a maximum degree in G[$V \backslash V'$]. Therefore vertex 3 is added to the ordering. There are edges $\{v_3, v_1\}$, $\{v_3, v_2\}$, $\{v_3, v_4\}$ and $\{v_3, v_5\}$. Incidence degree of 1,2,3 and 4 is increased to 1. A new degree list (1) is formed. Also, ideg is updated as the degree of 1,2,3 and 4 is decreased in G[$V \backslash V'$]. HEAD, NEXT, and PREVIOUS get updated to traverse the new list. The updated data structures are shown in figure 3.15.

All the unordered vertices have the same incidence degree. Also, they have the same degree in ideg. In this case, vertex 1 is selected as it is the first element of the list. Vertex 1 is added to the ordering. The incidence degree of vertex 4 is increased as there is an edge between vertex 1 and vertex 4. Also, the degree of vertex 4 is decreased in ideg. The

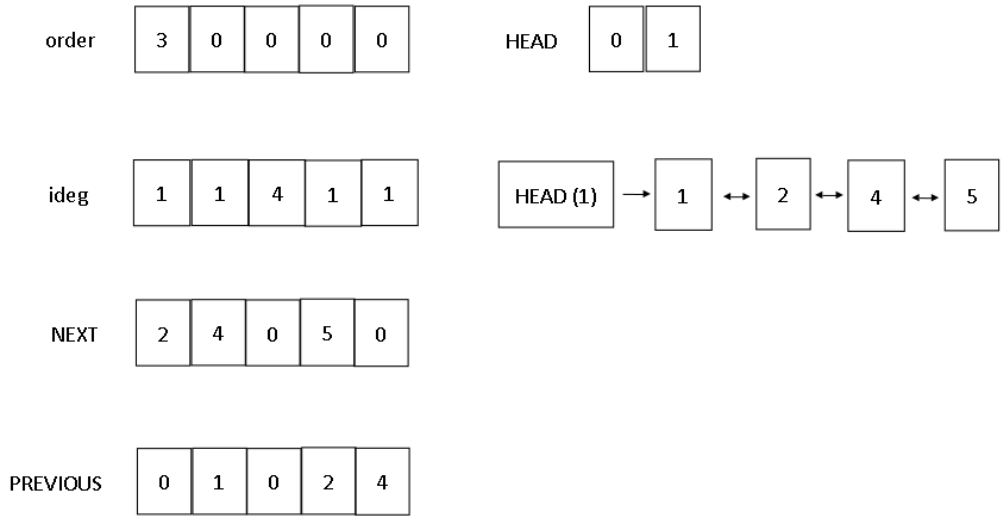

Figure 3.15: Data Structures after first iteration (IDO)

updated data structures are in figure 3.16.

In the third iteration, there is two incidence degree list. The maximum incidence degree list is 2. HEAD(2) = 4. Vertex 4 is added to the ordering. Incidence degree and degree at ideg of vertex 1 is updated as vertex 1 has an edge with vertex 4. The updated data structures are shown in figure 3.17.

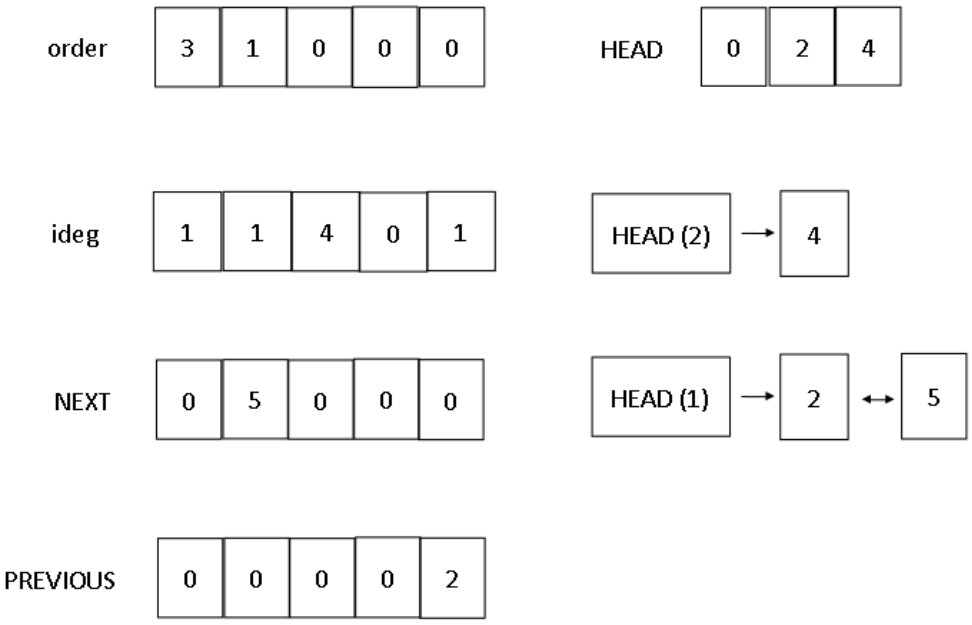

Figure 3.16: Data Structures after second iteration (IDO)

Now there are two vertices 2 and 5 in the unordered set. Both of them have same incidence degree as well as same degree in the ideg. HEAD(1) = 2 is processed first. Vertex 2 is added to the ordering. As vertex 2 and vertex 5 has an edge, incidence degree and degree at ideg is updated for vertex 5. The updated data structures are shown in figure 3.18.

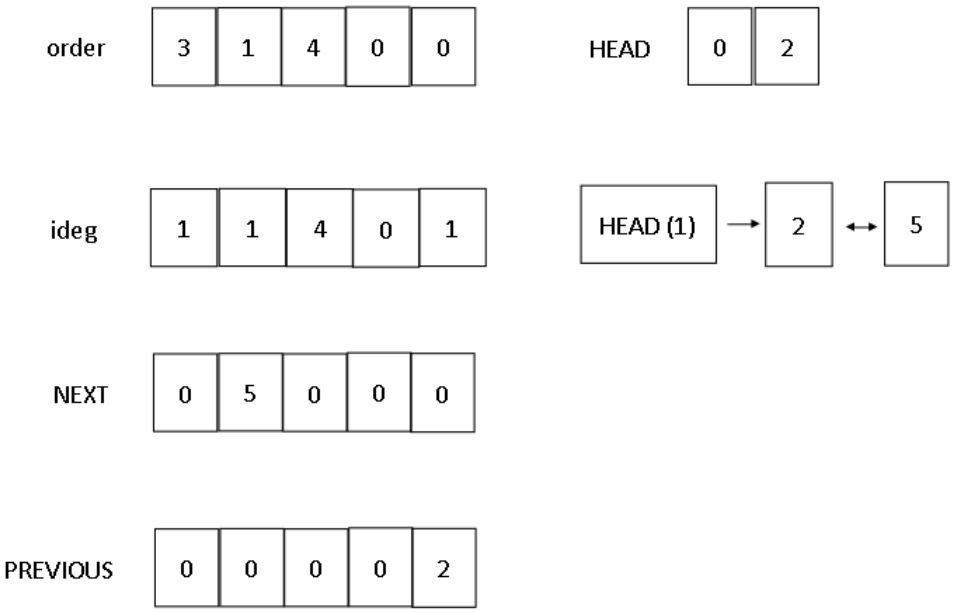

Figure 3.17: Data Structures after third iteration (IDO)

In the end, there is only one vertex remaining in the unordered set, which is vertex 5. Vertex 5 is added in the last position of order array. The final ordering is shown in figure 3.19.

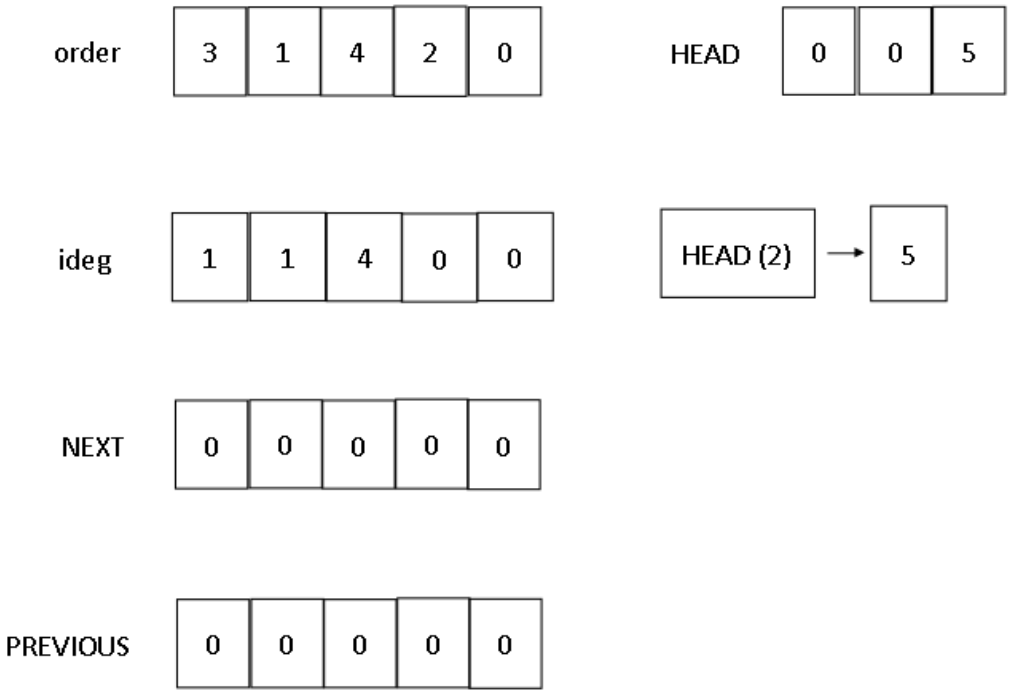

Figure 3.18: Data Structures after fourth iteration (IDO)

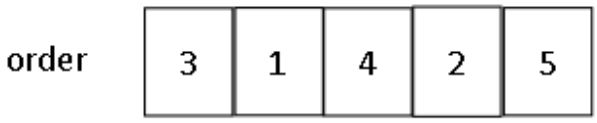

Figure 3.19: Final ordering of vertices (IDO)

3.2.3 Largest First Ordering (LFO)

Largest first ordering is the straightforward ordering algorithm. In LFO, vertices V = $\{v_1, v_2, ..., v_n\}$ are ordered in decreasing order. The algorithm [15] of LFO is listed in figure 3.20.

LFO gets the sparsity pattern *S(A)* of the matrix A as input and returns *order* array, which stores the ordering of vertices returned by LFO.

In LFO, vertex with the maximum degree is stored in the first position of order array. For that, lfindex is initialized to 1 in the algorithm in figure 3.20. j is an array consisting of all the column vertices. Loop in line 3 executes until all the column vertices are traversed.

```
LFO (S(A), order)
1       lfindex ← 1
2       j ← {1,2,....,n}
3       while j ≠ ∅
4           let l ∈ j be such that d_j(l) is maximum
5           order(lfindex) ← l
6           lfindex ← lfindex + 1
7           j ← j\ {l}
```

Figure 3.20: LFO Algorithm

LFO processes the vertex with a maximum degree at first and assigns that vertex to the first position of order array. Then lfindex gets updated by incremented by 1. Line 7 removes the ordered vertex from the unordered set of vertices. While loop in line 3 continues until all the vertices assigned in the order array. This algorithm is for column vertices. We have adopted this algorithm for row vertices.

Data structures after the initialization of matrix A for LFO is in figure 3.21.

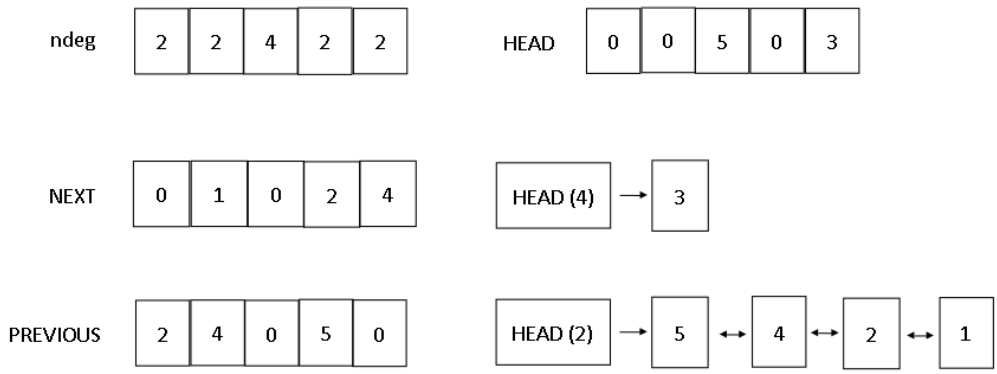

Figure 3.21: Data Structures after initialization of matrix A (LFO)

Initially, there are two-degree lists (2) and (4). Degree 4 is the largest one. HEAD(4) = 3. So 3 will be added at the beginning of the order array. In LFO, there will be no degree update of the remaining vertices after each iteration. Only the recently added vertex will be removed from the list. The updated data structures after iteration 1 are in figure 3.22.

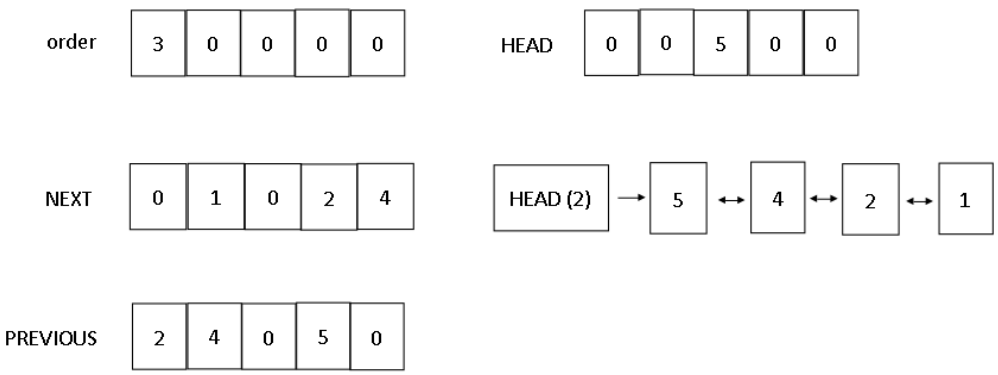

Figure 3.22: Data Structures after first iteration (LFO)

In the second iteration, there is only one-degree list (2). HEAD (2) = 5. So vertex 5 is removed from the unordered set and added to the order array. Data Structures after the second iteration is listed in figure 3.23.

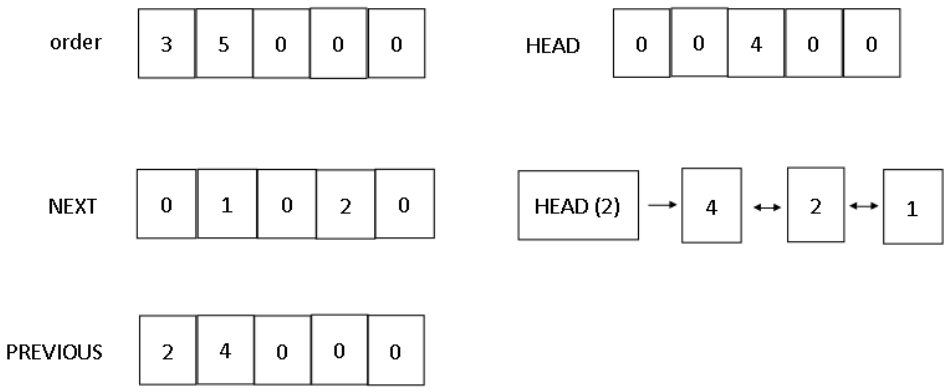

Figure 3.23: Data Structures after second iteration (LFO)

In the third iteration vertex, 4 is added to the ordering. There will be no changes in the degree lists. The updated data structures are represented in figure 3.24.

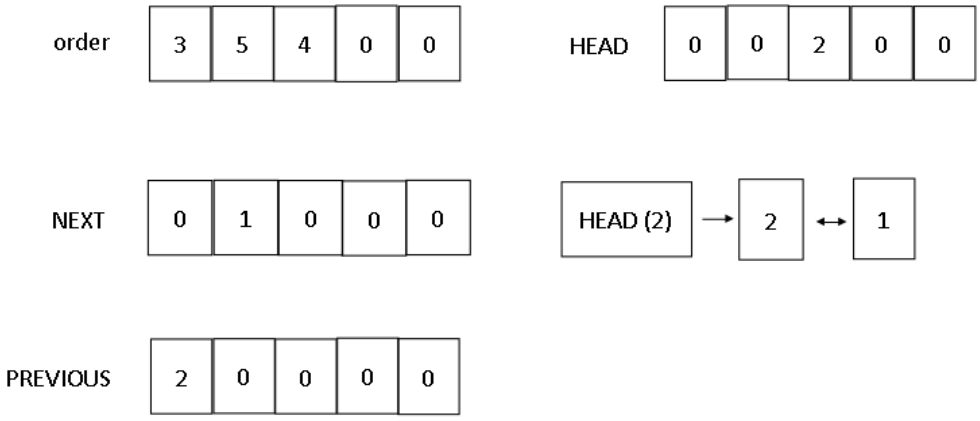

Figure 3.24: Data Structures after third iteration (LFO)

In the fourth iteration, there is an only one-degree list (2). HEAD(2) is vertex 2. Therefore vertex 2 is added to the ordering. The updated data structures are represented in figure 3.25.

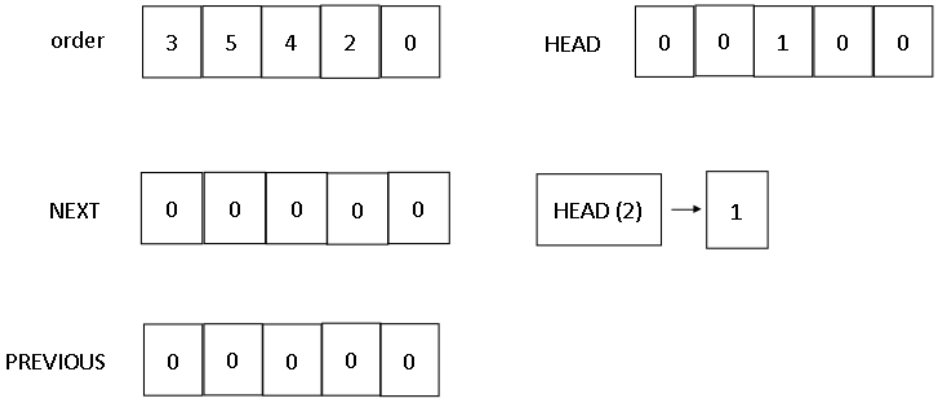

Figure 3.25: Data Structures after fourth iteration (LFO)

Finally, there is one vertex left, which is vertex 1. This vertex is added to the ordering. The final ordering is shown in figure 3.26.

Figure 3.26: Final ordering of vertices (LFO)

Chapter 4

Efficient Implementation of Coloring Algorithm

In this chapter, we describe the star bi-coloring algorithm we developed. The proposed algorithm is developed based on complete direct cover [17]. The star bi-coloring algorithm we propose determines non-zero entries of sparse matrices by forming groups of vertices representing rows and columns. The collection of vertices representing rows that are not connected by length-2 path can be grouped and known as row groups. In the same way, column groups can be formed. In a row, if there are more than one non-zero element in different columns, then one row group or color cannot determine all the non-zeroes. One group can determine one non-zero element in a row. Remaining non-zero elements need to be determined by either a separate row group or a column group. The same condition works for the column. If there are more than one non-zero elements in a column in different rows, then one column group or color can not determine all the non-zero elements in that column. One non-zero will be determined by a column group. The rest of the non-zeroes need to be determined by different column groups or row groups.

Different steps of our coloring algorithm are discussed in this chapter. Then all the steps are combined to state our algorithm. In the end, coloring by the algorithm is verified.

Matrix A from equation (3.3) and the graph representation of matrix A from figure 3.1 will be used as a reference matrix for different illustrations.

4.1 Background

In the earlier research works on partitioning algorithms mainly focused on unidirectional partitioning. Graph coloring can be used to solve the partitioning problem was first proposed by Coleman and More [3]. They developed their algorithm for the column intersection graph. They worked with different partitioning algorithms, such as SLO, LFO, and IDO. Hasan [15] proposed Recursive Largest First (RLF)ordering method for star bicoloring. Khan [22] also worked on unidirectional partitioning and implemented a branch and bound type exact coloring algorithm. For optimal partitioning, Suny [24] proposed a new algorithm that combines existing exact and heuristic algorithms for optimal partitioning.

All the above research works were done focusing on the column of the matrices. Only column groups were constructed to partition the matrix. The bidirectional approach can determine more non-zero entries of a matrix than a unidirectional approach. In [13], Mini goyal worked on bidirectional partitioning and showed that it has a better result than unidirectional partitioning. Anik Saha [23] also worked on Bi-directional determination of sparse Jacobian matrices. He proposed heuristic and iterative algorithms to determine the non-zero entries of a sparse Jacobian matrix. Juedes and Jones proposed a new star bicoloring algorithm in [20], which is known as approximate star bi-coloring. In this thesis, we are proposing a star bi-coloring algorithm inspired by the complete direct cover algorithm proposed in [17]. We can represent a matrix as a bipartite graph by constructing a graph putting rows in one vertex set and putting columns in separate vertex sets. In our algorithm, the degree of both row and column vertices are computed first. We start processing with vertex having a maximum degree. This can be row vertex or column vertex. Vertices with distance-2 neighbors are grouped separately. Vertices that are not in the distance-2 neighbor list can be grouped together.

4.2 Degree Calculation

Calculating the degree of vertices is an important part of our algorithm. The vertex with maximum degree starts a group, and the rest of the process continues. We calculated the degree for both column and row vertices. We wrote a procedure called computedegreeforbipartite(), and that procedure is listed in figure 4.1.

```
computedegreeforbipatite (S(A))

1   for every vertex v_i representing column vertex where i = 1,2,....,N
2       deg(v_i) ← number of row vertices connected to v_i
3   maxDegree_col ← deg(v_1)
4    maxCol ← v_1
5   for i = 2 to N do
6      if deg(v_i) >maxDegree_col then
7          maxDegree_col ← deg(v_i)
8          maxCol ← v_i
9   for every vertex u_i representing row vertex where i = 1,2,....,M
10      deg(u_i) ← number of column vertices connected to v_i
11  maxDegree_row ← deg(u_1)
12  maxRow ← u_1
13  for i = 2 to M do
14      if deg(u_i) >maxDegree_row then
15          maxDegree_row ← deg(u_i)
16          maxRow ← u_i
```

Figure 4.1: Algorithm for Degree Calculation

In the algorithm, $deg(v_i)$ denotes the degree of a vertex v_i. Degree of a vertex is equal to the number of adjacent nodes of that vertex. Degree of column vertices are computed using the loop in line 1. The variable maxDegree_col stores the maximum degree of column vertices, and maxCol denotes the column number, which has a maximum degree. maxDegree_col and maxCol are evaluated using the loop in line 5. maxDegree_col is initialized with the degree of first column vertex and maxcol is initialized with column 1. Line 6 to line 8 compares the degree of other column vertices with maxdegree_col and assigns the maximum degree in maxdegree_col. Also maxcol gets the column vertex having maximum degree. The variables maxDegree_row and maxRow works in same way for row vertices.

The degree of vertices of matrix A is shown in figure 4.2.

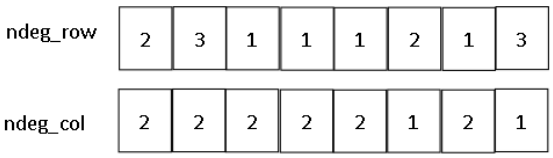

Figure 4.2: Degree of vertices of Matrix A from equation 3.3

The maximum degree among row vertices is 3, and the maximum degree among column vertices is 2.

4.3 Distance-2 Neighbor List Calculation

To be in the same group, the vertices need to be from the same bipartition, and there can not be a length 2 path among the vertices. For that, the distance-2 neighbor list calculation is a significant operation before forming the group.

The algorithm for distance-2 neighbor list calculation for columns is represented in figure 4.3.

The variable processingCol has the column vertex for which we need to find the distance-2 neighbor list. Line 3 retrieves the list of rows connected to processingCol. The list retrieved in line 3 is considered as distance-1_neighbor_list. The loop in line 4 iterates for

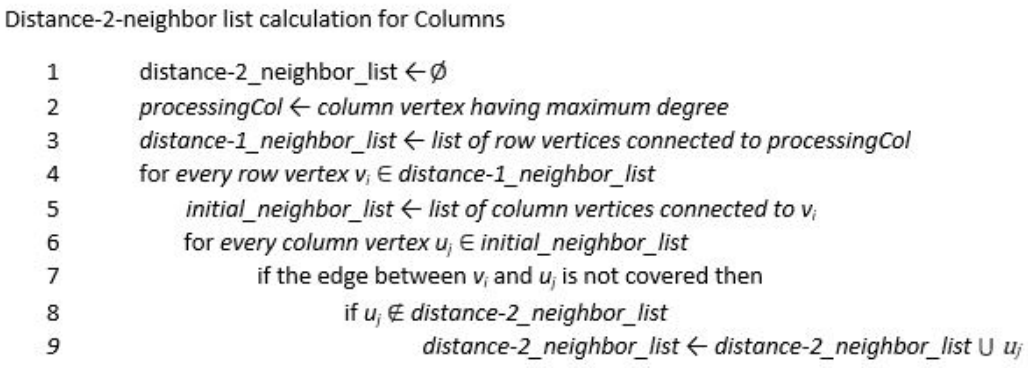

Figure 4.3: Algorithm for distance-2 neighbor list calculation for columns

all the row vertices of distance-1_neighbor_list. Line 5 retrieves the list of adjacent column vertices for processing row vertex. A column vertex is added to the distance-2_neighbor_list if the edge between processing row vertex and processing column vertex is not covered and the processing column is not already in the distance-2_neighbor_list.

The algorithm for distance-2 neighbor list calculation for rows is represented in figure 4.4.

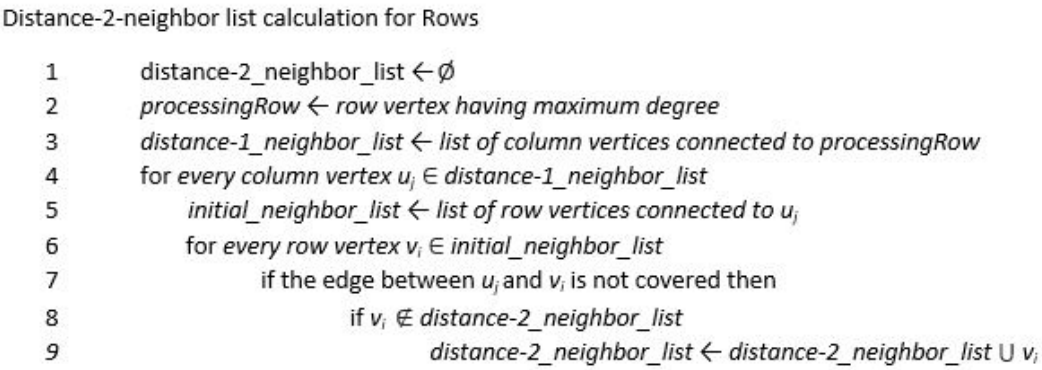

Figure 4.4: Algorithm for distance-2 neighbor list calculation for rows

The algorithm for distance-2 neighbor list calculation for rows works in the same manner as the algorithm for columns.

Now We will illustrate the distance-2 neighbor calculation using the graph in figure 3.1. We will also need to use the CSR and CSC representation of matrix A in figure 3.2 and

figure 3.3.

Suppose we would like to find out the distance-2 neighbors of column 3. Rows connected to column 3 = colptr(3) : colptr (4) - 1 = 4 : 5. Now, rowind (4:5) = $\{1,4\}$. So, row 1 and row 4 are connected to column 3.

Now we need to find out the columns connected to row 1 and row 4. Columns connected to row 1 = rowptr(1) : rowptr (2) - 1 = 1:2. Now, colind (1:2) = $\{1,3\}$. Therefore, column 1 is a distance-2 neighbor of column 3. As a result, column 3 and column 1 can not be grouped together.

Again, columns connected to row 4 = rowptr(4) : rowptr (5) -1 = 6 : 6. Now, colind(6) = 3. So the final distance-2 neighbor list of column 3 is = $\{column1\}$. So column 1 can not be grouped with column 3. Other columns can be grouped with column 3.

Now we will discuss about how to determine the distance-2 neighbor list of a row vertex. Suppose we are processing row 8. Columns connected to row 8 = rowptr (8) : rowptr (9) - 1 = 12 : 14. colind (12:14) = $\{2,7,8\}$. Therefore, row 8 is connected to column 2, column 7 and column 8.

Then we need to find out the rows connected to column 2, column 7 and column 8. Rows connected to column 2 = colptr (2) : colptr (3) - 1 = 3 : 4. Now, rowind (3:4) = $\{2,8\}$. As a result, column 2 is distance-2 neighbor of column 8. Rows connected to column 7 = colptr (7) : colptr (8) - 1 = 12 : 12. Now, rowind(12) = 2. We again find that column 2 is distance-2 neighbor of column 8. Rows connected to column 8 = colptr(8) : colptr(9) - 1 = 13:14. Now, rowind (13:14) = $\{8\}$. The final distance-2 neighbor list of row 8 = $\{2\}$. For that, row 2 can not be grouped together with row 8.

4.4 Formation of the Groups

After determining the distance-2 neighbor list of a vertex, any vertices which are not in the distance-2 neighbor list can be added to the current group. When a vertex is added to the group, then the distance-2 neighbor list of newly added vertex needs to be merged with

the previous distance-2 neighbor list. We determined the distance-2 neighbor list of row 8 in the previous section. The list is $\{2\}$. Therefore, row 2 can not be grouped together with row 8. Adding row 1 in the group = $\{1,8\}$. Distance-2 neighbor list of row 1 = $\{4,6\}$. Updated distance-2 neighbor list = $\{2\} \cup \{4,6\} = \{2,4,6\}$. So row 2, row 4 and row 6 can not be added to current group. Row 3 is added to the group = $\{1,3,8\}$. Distance-2 neighbor list of row 3 = row 2. Distance-2 neighbor list remains same. Row 5 is added to the group = $\{1,3,5,8\}$. Distance-2 neighbor list of row 5 = row 7. Updated distance-2 neighbor list = $\{2,4,6,7\}$. So the first row group is $\{1,3,5,8\}$.

In the same procedure, column groups can be constructed.

4.5 Updating Degrees

The proposed algorithm iterates, while edgecount is less than E. Here E, is the total number of edges, which is equal to the total number of non-zero entries. Algorithm starts with edgecount = 0. After each iteration, the edgecount gets updated.

In the last section, we formed a row group, $\{1,3,5,8\}$. All the edges connected to the group members are removed from the graph. So, all the group members' degree will be 0, and the degree of columns which are connected to the rows of the groups are updated. The degree of both row and column vertices are decreased. Edgecount increases by the number of edges removed from the graph in that iteration. 7 edges are removed from the graph after forming the row group $\{1,3,5,8\}$. Edgecount is 7 after the first iteration. Degree of row 1,2,5,8 are 0. The degree of column 1,3,4, 5,7,8, is reduced as they have edges with the rows belonging to the group.

After the degree of vertices is updated, vertex with the maximum degree will be selected to start a group for the next iteration.

4.6 Partitioning Algorithms for Row

We have implemented the proposed coloring algorithm using DSJM [16]. Partitioning algorithms such as SLO, LFO, and IDO are defined in DSJM. However, DSJM was implemented using unidirectional partitioning. Basically, all the partitioning algorithms are implemented based on the column intersection graph in DSJM. Executing partition algorithms returns the ordering of column vertices. As we are considering bidirectional partitioning, ordering of row vertices is also needed. We transposed the given matrix. After the transpose operation, the column of matrix A becomes the row of A^T, and the row of matrix A becomes the column of A^T. Then we wrote the ordering functions such as SLO, LFO, and IDO for row by adopting the ordering functions defined in DSJM for the column. We used A^T in the ordering algorithms for the row to get the ordering of row vertices.

4.6.1 Transpose of Matrix

Both square and non-square matrix can be transposed using the algorithm in figure 4.5.

transpose(A)

1 Suppose the matrix holding the transpose of A is A^T
2 Assign the row data of A into the column data structures of A^T
3 Assign the column data of A into the row data structures of A^T

Figure 4.5: Algorithm for making transpose of a matrix

Transpose of a matrix converts the rows into columns and columns into rows. The row data of matrix A are stored in rowptr and colind using compressed sparse row (CSR). The column data of matrix A are stored in colptr and rowind using compressed sparse column (CSC). We have taken four new arrays transrowptr, transcolptr, transrowind and transcolind to store the transpose of matrix A^T. We assigned row information of matrix A to the column data structure of A^T. For example, we assigned rowptr values in transcolptr. After performing this algorithm, we got the transpose of the input matrix A. Using these new arrays SLO, LFO, and IDO are executed, which returned the ordering of row vertices.

4.7 Coloring Algorithm

In this section, we are merging all the procedures stated above to derive the coloring algorithm. This coloring algorithm determines all the non-zero entries of a sparse matrix in a bi-directional manner. Figure 4.6 shows the coloring algorithm.

```
coloring algortihm
1     edgecount ← 0
2     k ← 0
3     while edgecount < E do
4         k ← k + 1
5         if maxDegree_col ≥ maxDegree_row then
6             start a group_k with maxcol
7             find distance-2 neighbor list of maxcol
8             for j = 1 to N do
9                 add column_j in the group_k if it is not in distance-2 neighbor list
10                update distance-2 neighbor list
11        if maxDegree_row > maxDegree_col then
12            start a group_k with maxrow
13            find distance-2 neighbor list of maxrow
14            for i = 1 to M do
15                add row_i in the group_k if it is not in distance-2 neighbor list
16                update distance-2 neighbor list
17        Let E_k be the set of edges which has end points in group_k
18        edgecount ← edgecount + |E_k|
```

Figure 4.6: Coloring Algorithm

In section 4.2, we have discussed degree calculation. Degrees of all the vertices are calculated using computedegreeforbipartite(). This function also calculates the maxDegree_row, maxrow, maxDegree_col and maxcol. Before executing the algorithm, these values are computed. Here, edgecount is the loop control variable. Initially, the value of the edgecount is 0. In each iteration, the value of edgecount increases based on the number of the edge covered by the vertices in $group_k$. E is the total number of edges. The total number of edges is equal to the total number of non-zeroes in the matrix. The grouping procedure starts with the vertex having a maximum degree. Step 5 checks whether a column vertex has a maximum degree or not. The same checking is executed for row vertex in step 11. If both row and column vertex have the same maximum degree, we start the grouping procedure with column vertex. After starting a group with the maximum degree vertex, the distance-2 neighbor list is calculated for processing vertex. Distance-2 neighbors can not

be added to the group. Vertices that are from the same bipartition and which are not in the distance-2 neighbor list can be added in the group. Using the loop of step 8 and step 14, remaining vertices from the same bipartition are traversed. Whenever a vertex is added to the group, the distance-2 neighbor list is updated. After forming the group, the edgecount is updated by adding the number of edges that have endpoints to the vertices of the group. These edges are considered to be covered by the vertices of the group and need to be left out from the degree calculation of the next iteration. Step 8 and step 14 denote the natural order of processing. We have also used SLO, LFO, and IDO ordering in these steps and found different results based on a number of colors and time taken.

We are going to explain the algorithm using an example matrix A from equation (3.3) and graph representation form figure 3.1.

There are 14 non-zeroes in matrix A. So E = 14. At the beginning edgecount= 0. So, edgecount is less than E. The degree of the vertices are shown in figure 4.2. From the figure 4.2 we can deduce that, maxDegree_row = 3, maxrow = 2, maxDegree_col = 2 and maxcol = 1.

As maxDegree_row is greater than maxDegree_col, the first group will be formed using row vertices. First member of the group is row 2. Distance-2 neighbor list of row 2 = $\{3,8\}$. Therefore, row 3 and row 8 can not be added to the current group. Then row 1 is added to the group = $\{1,2\}$. Updating distance-2 neighbor list as row 2 is a new member of the group = $\{3,4,6,8\}$. Adding row 5 in the group = $\{1,2,5\}$. updating distance-2 neighbor list = $\{3,4,6,7,8\}$. Let us assign color blue to group 1.

Figure 4.7 shows the current state of the graph after removing the edges which have endpoints at the grouped vertices.

After the first iteration, the edgecount is 6. The current state of the degree of vertices is listed in figure 4.8.

maxDegree_row = 3, maxrow = 8, maxDegree_col = 1 and maxcol = 1.

In the second iteration, edgecount = 6 is less than E (=14). maxDegree_row is greater

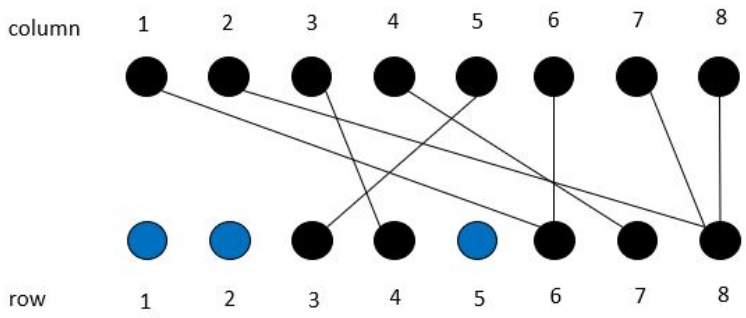

Figure 4.7: State of the graph after first iteration

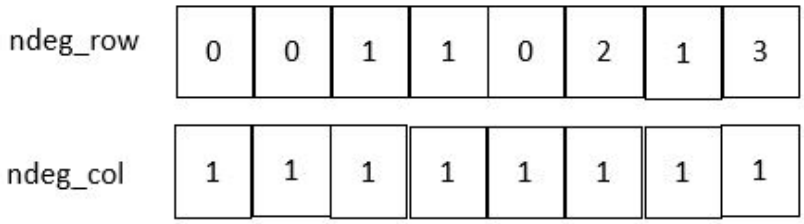

Figure 4.8: Degree of vertices after first iteration

than maxDegree_col. Starting a new group with row 8. Distance-2 neighbor list of row 8 = $\{\varnothing\}$. Adding row 3 in group = $\{3,8\}$. Updating distance-2 neighbor list = $\{\varnothing\}$. Adding row 4 in the group = $\{3,4,8\}$. Distance-2 neighbor list is still empty. Adding row 6 in the group = $\{3,4,6,8\}$. Distance-2 neighbor list is still empty after adding row 6. Adding row 7 in the group = $\{3,4,6,7,8\}$. All the row vertices are grouped which makes the edgecount to 14. As the edgecount becomes equal to E, the while loop terminates. All the edges are covered. That means all the non-zeroes of matrix A are determined. Suppose we assigning color green to the new group. The final state of graph with all edges is shown in figure 4.9.

4.8 Verification of the Coloring

We verified the coloring of our proposed algorithm by using vector-matrix product. Group1 vector is [1 1 0 0 1 0 0 0]. The product of group1 vector with matrix A is represented

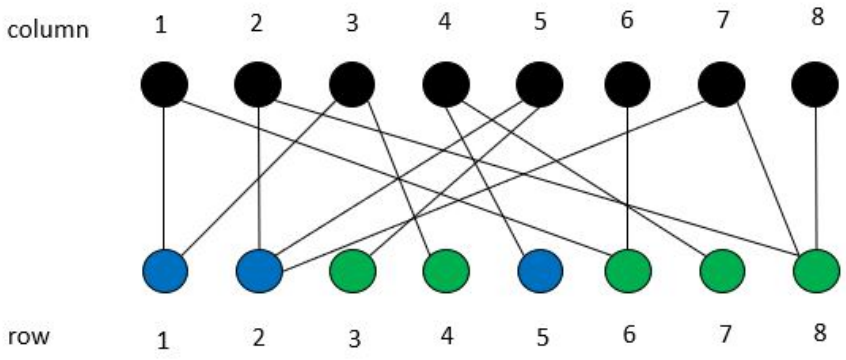

Figure 4.9: Final state of the graph after coloring

in figure 4.10.

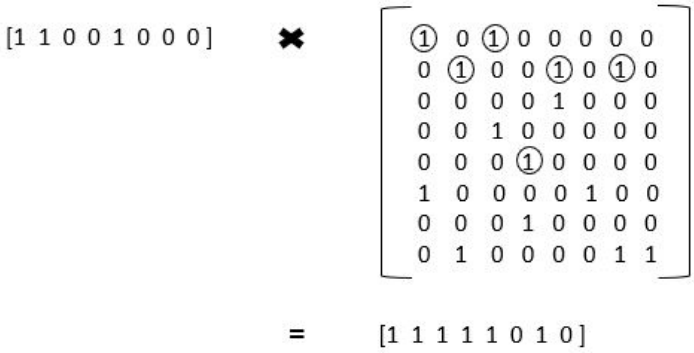

Figure 4.10: Multiplication result of group1 vector with matrix A

We assumed that all the non-zero values of matrix A are 1. If the multiplication result has any value greater than 1, that denotes non-zeroes can not be determined in that row. But in the result above, all the values are 0 or 1. The encircled 1 values are determined by group 1.

We are assigning 0 to the determined values of matrix A. Then multiplying A with group 2 vector = [0 0 1 1 0 1 1 1]. The product of the group 2 vector with matrix A is represented in 4.11.

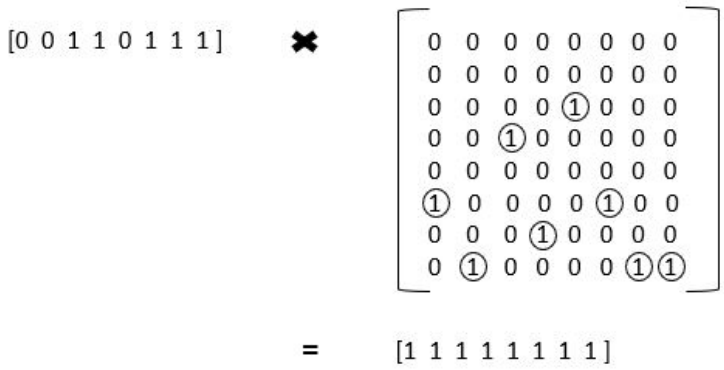

Figure 4.11: Multiplication result of group 2 vector with matrix A

All the remaining non-zero elements are determined by group 2. There are no non-zero entries left undetermined.

Chapter 5

Numerical Experiments

In this chapter, we will discuss the numerical experiments we carried out and their results. In the beginning, we will explain about test data set and test environment. Then, we will present the results of numerical experiments. In the end, we will discuss the findings from the result.

5.1 Test Data Sets

We have tested our algorithm on different sparse matrices. The sparse matrices are collected from Matrix Market Collection [1]. The test matrices are general matrices. In general, these matrices are defined using three attributes-

- **m**: m represents the number of rows of the matrix.
- **n**: n denotes the number of columns of the matrix.
- **nnz**: nnz indicates the number of non-zero elements in the matrix.

5.2 Test Environment

All numerical experiments are carried out in a computer located at the University of Lethbridge. The details of the device are listed in table 5.1.

Table 5.1: Details of the test environment

Processor	Operating System	Cache (L2)	RAM	No. of Core
Intel Core i7 4770 CPU 3.40 GHz	Linux	256 KB	8 GB	4

5.3 Test Results

We have used 2 sets of data for our experiment. Matrices in table 5.2 (data set 1) are used to compare the result of our algorithm with [21]. These are large matrices. Other matrices in table 5.3 (data set 2) are used in experiments to compare our algorithm with [20]. These matrices are small matrices. The experimental results of data set 2 are also compared with [23].

Table 5.2: Matrix statistics for data set 1

Matrix	M	N	NNZ	Row Max	Col Max	Row Average	Col Average
af23560	23560	23560	484256	21	21	20.55	20.55
cage11	39082	39082	559722	31	31	14.32	14.32
cage12	130228	130228	2032536	33	33	15.61	15.61
e30r2000	9661	9661	306356	62	62	31.71	31.71
e40r0100	17281	17281	553562	62	62	32.03	32.03
lhr10	10672	10672	232633	63	36	21.80	21.80
lhr14	14270	14270	307858	63	36	21.57	21.57
lhr34	35152	35152	764014	63	36	21.73	21.73
lp_cre_a	3516	7248	18168	360	14	5.17	2.51
lp_cre_b	9648	77137	260785	844	14	27.03	3.38
lp_cre_d	8926	73948	246614	808	13	27.63	3.33
lp_dfl001	6071	12230	35632	228	14	5.87	2.91
lp_ken_11	14694	21349	49058	122	3	3.34	2.30
lp_ken_13	28632	42659	97246	170	3	3.40	2.28
lp_ken_18	105127	154699	358171	325	3	3.41	2.32
lp_maros_r7	3136	9408	144848	48	46	46.19	15.40
lp_pds_10	16558	49932	107605	96	3	6.50	2.16
lp_pds_20	33874	108175	232647	96	3	6.87	2.15
lp_stocfor3	16675	23541	76473	15	18	4.59	3.25

Here, matrix= represents the name of the matrix, M= number of rows in the matrix, N=

number of columns in the matrix, Row Max= maximum degree among row vertices, Col Max = maximum degree among column vertices, Row Average = average degree of row vertices, Col Average = average degree of column vertices.

Table 5.3: Matrix statistics for data set 2

Matrix	M	N	NNZ	Row Max	Col Max	Row Average	Col Average
abb313	313	176	1557	6	26	4.97	8.85
ash219	219	85	438	2	9	2	5.15
ash331	331	104	662	2	12	2	6.37
ash608	608	188	1216	2	12	2	6.47
ash958	958	292	1916	2	13	2	6.56
bp0	822	822	3276	266	20	3.99	3.99
bp200	822	822	3802	283	21	4.63	4.63
bp400	822	822	4028	295	21	4.90	4.90
bp600	822	822	4172	302	21	5.08	5.08
bp800	822	822	4534	304	21	5.52	5.52
bp1000	822	822	4661	308	21	5.67	5.67
bp1200	822	822	4726	311	21	5.75	5.75
bp1400	822	822	4790	311	21	5.83	5.83
bp1600	822	822	4841	22	22	21	0
fs_541_1	541	541	4285	11	541	7.92	7.92
fs_541_2	541	541	4285	11	541	7.92	7.92
ibm32	32	32	126	8	7	3.94	3.94
shl0	663	663	1687	422	4	2.54	2.54
shl200	663	663	1726	440	4	2.60	2.60
shl400	663	663	1712	426	4	2.58	2.58
str0	363	363	2454	34	34	6.76	6.76
str200	363	363	3068	30	26	8.45	8.45
str400	363	363	3157	33	34	8.70	8.70
str600	363	363	3279	33	34	9.03	9.03
will57	57	57	281	11	11	4.93	4.93
will199	199	199	701	6	9	3.52	3.52

We have used natural order, SLO order, LFO Order, and IDO order in the coloring algorithm. A comparison of the results of using different ordering on data set 1 is listed in 5.4.

Table 5.4: Comparison of proposed coloring algorithm for Natural order, SLO order, LFO order and IDO order for data set 1

Matrix	Natural Order			LFO			SLO			IDO		
	RG	CG	T	RG	CG	T	RG	CG	T	RG	CG	T
af23560	0	48	36.91	0	64	36.67	0	45	37.53	0	49	36.54
cage11	0	79	48.83	0	77	41.56	0	62	41.50	0	64	41.85
cage12	0	97	652.51	0	89	546.93	0	69	599.84	0	70	548.21
e30r2000	0	68	18.04	0	89	17.37	0	69	17.65	0	71	17.73
e40r0100	0	95	57.02	0	91	56.17	0	71	57.54	0	71	58.46
lhr10	64	0	11.24	65	0	11.73	35	63	31.07	34	63	31.19
lhr14	64	0	19.15	66	0	20.72	35	64	54.55	34	64	55.02
lhr34	64	0	117.69	65	0	128.61	35	63	333.69	34	64	340.13
lp_cre_a	15	0	0.03	15	0	0.03	14	0	0.04	14	0	0.05
lp_cre_b	16	0	0.90	18	0	0.79	17	0	0.97	16	0	1.18
lp_cre_d	15	0	0.73	18	0	0.68	15	0	0.87	15	0	1.04
lp_dfl001	15	0	0.10	15	0	0.10	14	0	0.11	14	0	0.22
lp_ken_11	5	0	0.28	5	0	0.25	4	0	0.26	4	0	0.28
lp_ken_13	4	0	1.08	6	0	0.92	4	0	1.05	4	0	1.15
lp_ken_18	5	0	14.21	5	0	13.19	5	0	13.50	4	0	14.31
lp_maros_r7	72	0	3.16	96	0	3.13	86	0	3.11	94	0	3.13
lp_pds_10	5	0	0.64	7	0	0.64	5	0	0.56	5	0	0.61
lp_pds_20	5	0	2.83	7	0	2.79	5	0	2.40	5	0	2.70
lp_stocfor3	0	15	1.29	0	17	1.41	0	15	1.22	0	16	1.30

Here, RG = Row Groups, CG = Column Groups, T = Time (in sec).

There are 19 matrices in data set 1. SLO requires a minimum number of colors (groups) to determine all the non-zeroes for 12 matrices. Natural order returned a minimum number of colors (groups) for 11 matrices. IDO returned a minimum number of colors (groups) for 10 matrices. LFO took less time to run the coloring algorithm in 10 instances. However, LFO always needed more colors (groups) to determine the non-zeroes in comparison to other ordering methods.

A comparison of the results of using different ordering on data set 2 is listed in 5.5.

Table 5.5: Comparison of proposed coloring algorithm for Natural order, SLO order, LFO order and IDO order for data set 2

Matrix	Natural Order			LFO			SLO			IDO		
	RG	CG	T	RG	CG	T	RG	CG	T	RG	CG	T
abb313	0	11	0.0	0	13	0.0	0	10	0.0	0	10	0.0
ash219	0	5	0.0	0	5	0.0	0	5	0.0	0	4	0.0
ash331	0	6	0.0	0	7	0.0	0	6	0.0	0	6	0.0
ash608	0	6	0.0	0	7	0.0	0	6	0.0	0	6	0.0
ash958	0	7	0.0	0	6	0.0	0	6	0.0	0	6	0.0
bp0	17	3	0.0	18	2	0.0	15	5	0.0	15	3	0.0
bp200	21	0	0.0	21	0	0.0	21	0	0.0	16	7	0.01
bp400	21	0	0.0	21	0	0.0	21	0	0.01	18	7	0.01
bp600	21	0	0.01	21	0	0.0	21	0	0.0	17	8	0.01
bp800	23	0	0.01	22	0	0.01	21	0	0.01	21	0	0.01
bp1000	23	0	0.0	25	0	0.0	22	0	0.00	22	0	0.01
bp1200	23	0	0.01	23	0	0.01	21	0	0.01	21	0	0.01
bp1400	23	0	0.00	24	0	0.0	21	0	0.0	21	0	0.01
bp1600	24	0	0.01	24	0	0.01	21	0	0.01	21	0	0.01
fs_541_1	0	15	0.0	0	18	0.0	0	13	0.0	0	13	0.0
fs_541_2	0	15	0.0	0	18	0.0	0	13	0.0	0	13	0.0
ibm32	1	7	0.0	1	8	0.0	1	7	0.0	1	7	0.0
shl0	4	0	0.0	5	0	0.0	4	0	0.0	4	0	0.0
shl200	4	0	0.0	5	0	0.0	4	0	0.0	4	0	0.0
shl400	4	0	0.0	5	0	0.0	4	0	0.0	4	0	0.0
str0	4	35	0.0	4	34	0.0	4	30	0.0	6	25	0.0
str200	33	0	0.0	32	0	0.0	32	0	0.0	12	25	0.0
str400	18	23	0.0	36	1	0.0	35	1	0.0	10	25	0.0
str600	23	19	0.0	37	1	0.0	35	1	0.0	12	27	0.0
will57	7	10	0.0	6	10	0.0	11	3	0.0	11	2	0.0
will199	1	8	0.0	0	9	0.0	0	7	0.0	0	7	0.0

There are 26 matrices in data set 2. SLO and IDO require a minimum number of colors (groups) to determine all the non-zeroes for 21 matrices. Natural order returned a minimum number of colors (groups) for 8 matrices. LFO returned a minimum number of colors (groups) for 4 matrices.

We can observe that SLO and IDO produce a better result than Natural order and LFO based on the experimental results from table 5.4 and table 5.5. LFO takes less running time

in comparison to SLO, IDO, and natural order, but LFO requires more groups to determine all the non-zeroes in comparison to other ordering methods.

5.3.1 Comparison with [Juedes and Jones, 2019]

In [21], Juedes and Jones experimented on different greedy algorithms of star bi-coloring. These algorithms are-

- Approximate Star Bi-Coloring (ASBC): ASBC is an approximation algorithm that uses distance-2 independent sets of vertices in the bipartite graph [20].
- Max-Neighborhood (MN): MN selects the largest neighborhood among two alternatives, independent sets to form the bipartition [21].
- Max-Ratio (MR): MR_c where $0 < c < 1$, forms bipartition showing the maximum ratio of the size of its neighborhood ($|\lceil(V_i)|$) with respect to the size of the opposite bipartition ($|V_{not-i}|$) [21].
- Local-Search-K: LS-K for some fixed $k > 1$, forms at most k distance-2 independent sets by generating all sequences of distance-2 independent sets from one of the two bipartitions and selecting the sequence that deletes the maximum number of edges [21].

Table 5.6 shows the result comparison of our coloring algorithm with [21]. In the table 5.6, PCA means proposed coloring algorithm. The experimental results of our algorithm are listed in column PCA.

Table 5.6: Comparison of proposed coloring algorithm with [Juedes and Jones, 2019] [21]

Matrix	ASBC	MN	LS-2	LS-3	LS-4	LS-5	MR 0.5	MR 0.67	MR 0.75	PCA
af23560	52	36	36	36	36	36	36	36	36	45
cage11	82	68	68	68	68	68	68	68	68	62
cage12	91	72	72	72	72	72	72	72	72	69
e30r2000	81	72	72	72	72	72	72	72	72	68
e40r0100	92	83	83	83	83	83	83	83	83	71
lhr10	45	40	40	39	39	38	40	40	40	64
lhr14	45	40	39	39	39	38	40	40	40	64
lhr34	45	40	39	39	39	39	40	40	40	64
lp_cre_a	16	16	16	17	14	14	23	19	19	14
lp_cre_b	60	15	15	15	15	15	15	15	15	16
lp_cre_d	75	15	15	15	15	15	15	15	15	15
lp_dfl001	14	14	12	11	11	11	20	11	14	14
lp_ken_11	15	4	4	4	4	4	4	4	4	4
lp_ken_13	12	4	4	4	4	4	9	9	4	4
lp_ken_18	13	4	4	4	4	4	11	11	4	4
lp_maros_r7	144	70	70	70	88	70	70	70	70	72
lp_pds_10	29	6	6	6	6	6	6	6	6	5
lp_pds_20	34	6	6	6	6	6	6	6	6	5
lp_stocfor3	22	19	15	15	15	15	15	15	15	15

Here, we have enlisted the best result we get using natural order or SLO order or LFO order or IDO order. In many cases, our proposed algorithm gives better results than others. Our proposed coloring algorithm requires less colors to determine all non-zeroes for cage11, cage12, e30r200, e40r0100, *ip_cre_a*, *ip_pds*_10, *ip_pds*_20 compared to the algorithms of [21].

5.3.2 Comparison with [Juedes and Jones, 2011]

Juedes and Jones proposed a new star bi-coloring algorithm in [20], which is approximate star bi-coloring (ASBC). The authors have re-implemented minimum non-zero count ordering (MNCO) developed by Coleman and Verma [4]. MNCO partitions the sparse matrix into sub-matrices, which lead to either row-wise or column-wise coloring. There are

two versions of MNCO: direct and substitution. MNCO direct was re-implemented in [20]. Table 5.7 shows the result comparison of our coloring algorithm with ASBC and MNCO using data set 2.

Table 5.7: Comparison of proposed coloring algorithm with ASBC and MNCO direct

Matrix	M	N	NNZ	No. of cols ASBC	No. of cols MNCO direct	No. of cols PCA
abb313	313	176	1557	17	12	10
ash219	219	85	438	8	10	4
ash331	331	104	662	10	7	6
ash608	608	188	1216	11	11	6
ash958	958	292	1916	12	7	6
bp0	822	822	3276	16	15	18
bp200	822	822	3802	17	17	21
bp400	822	822	4028	19	19	21
bp600	822	822	4172	20	18	21
bp800	822	822	4534	22	21	21
bp1000	822	822	4661	22	21	22
bp1200	822	822	4726	21	22	21
bp1400	822	822	4790	23	22	21
bp1600	822	822	4841	22	22	21
fs_541_1	541	541	4285	18	14	13
fs_541_2	541	541	4285	18	14	13
ibm32	32	32	126	9	9	8
shl0	663	663	1687	7	5	4
shl200	663	663	1726	7	5	4
shl400	663	663	1712	6	5	4
str0	363	363	2454	25	27	31
str200	363	363	3068	31	28	32
str400	363	363	3157	36	30	35
str600	363	363	3279	36	30	36
will57	57	57	281	10	9	13
will199	199	199	701	9	6	7

Again, we have enlisted the best result we get using natural order or SLO order or LFO order or IDO order. In many cases, our proposed algorithm gives better results than others. Our proposed coloring algorithm requires less colors to determine all non-zeroes

of abb313, ash219, ash331, ash608, ash958, bp1400, bp1600, fs_541_1, fs_541_2, ibm32, shl0, shl200, shl400 compared to ASBC and MNCO direct.

5.3.3 Comparison with [Saha, 2015]

Saha proposed a heuristic approach for bi-directional partitioning. Table 5.8 shows the result comparison of our coloring algorithm with the heuristic coloring proposed in [23].

Table 5.8: Comparison of proposed coloring algorithm with [Saha, 2015]

Matrix	M	N	NNZ	No. of cols Saha	No. of cols PCA
abb313	313	176	1557	**10**	**10**
ash219	219	85	438	**4**	**4**
ash331	331	104	662	**6**	**6**
ash608	608	188	1216	**6**	**6**
ash958	958	292	1916	**6**	**6**
bp0	822	822	3276	**17**	18
bp200	822	822	3802	**19**	21
bp400	822	822	4028	**20**	21
bp600	822	822	4172	**19**	21
bp800	822	822	4534	22	**21**
bp1000	822	822	4661	**22**	**22**
bp1200	822	822	4726	**21**	**21**
bp1400	822	822	4790	**21**	**21**
bp1600	822	822	4841	**21**	**21**
fs_541_1	541	541	4285	**13**	**13**
fs_541_2	541	541	4285	**13**	**13**
ibm32	32	32	126	**7**	8
shl0	663	663	1687	**4**	**4**
shl200	663	663	1726	**4**	**4**
shl400	663	663	1712	**4**	**4**
str0	363	363	2454	**27**	31
str200	363	363	3068	**31**	32
str400	363	363	3157	36	**35**
will57	57	57	281	**7**	13
will199	199	199	701	9	**7**

Our proposed coloring algorithm requires fewer colors to determine all the non-zeroes

of bp800, str400, and will199. Both approaches require same amount of colors for abb313, ash219, ash331, ash608, ash958, bp1000, bp1200, bp1400, bp1600, fs_541_1, fs_541_2, shl0, shl200 and shl400 to determine all the non-zeroes. Our experimental results are almost similar to the results of [23].

Chapter 6

Conclusion and Future Work

In this thesis, we have proposed a star bi-coloring algorithm that ensures the determination of all the non-zero elements of a sparse matrix. With numerous experiments on a standard set of problems, we ensured that the proposed coloring algorithm works better than existing coloring algorithms [21] [20]. Apart from the natural order of traversing row and column, we used SLO order, LFO order, and IDO order. These different orders of traversals lead to distinct outcomes.

We have implemented the algorithm using DSJM. For that, an in-depth study of DSJM (Determine Sparse Jacobian Matrices) is required. The data structures and the algorithms for ordering and partitioning included in DSJM are studied thoroughly. DSJM is developed for unidirectional partitioning. We have added some procedures in DSJM, which facilitates it to work with bi-directional partitioning.

6.1 Future Works

Following are the future research directions-

- Our proposed coloring algorithm works efficiently for graphs represented using the general pattern of matrices. It needs to perform efficiently for all types of matrix representations of graphs, such as symmetric matrices.
- Apart from the natural order for traversing row and columns, we have experimented using the smallest last ordering (SLO), largest first ordering (LFO), and incidence degree ordering (IDO). In the future, other ordering techniques such as recursive

largest first partitioning (RLFPartition), saturation degree partitioning (SDPartition) can be used. These ordering may lead to better results.

- In the real world, problems are getting larger day by day. The parallel implementation of our proposed algorithm may lead to faster and better outcomes.

Bibliography

[1] The matrix market project. *https://math.nist.gov/MatrixMarket/*, Accessed: 2019-04-12.

[2] Power network patterns. *https://math.nist.gov/MatrixMarket/data/Harwell-Boeing/bcspwr/bcspwr03$_l$g.html*,*Accessed* : $2019-08-15$.

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

[4] Thomas F Coleman and A. Verma. The efficient computation of sparse jacobian matrices using automatic diferentiation. *SIAM J. Sci. Comput.*, 18(4):1210–1233, 1998.

[5] A. R. Curtis, M. J. D. Powell, and J. K. Reid. On the estimation of sparse jacobian matrices. *J. Inst. Math. Appl*, 13(1):117–120, 1974.

[6] I. S. Duff, R. G. Grimes, and J. G. Lewis. Sparse matrix test problems. *ACM Trans. Math. Softw*, 15(1):1–14, 1989.

[7] Guillaume Fertin, André Raspaud, and Bruce Reed. Star coloring of graphs. *Journal of Graph Theory*, 47 (3):163–182, 2004.

[8] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman Co, 1979.

[9] Daya Gaur, Shahadat Hossain, and Anik Saha. Determining sparse jacobian matrices using two-sided compression: An algorithm and lower bounds. In Jacques Bélair, Ian A. Frigaard, Herb Kunze, Roman Makarov, Roderick Melnik, and Raymond J. Spiteri, editors, *Mathematical and Computational Approaches in Advancing Modern Science and Engineering*, pages 425–434. Springer International Publishing, 2016.

[10] A. H. Gebremedhin, F. Manne, and A. Pothen. What color is your jacobian? *graph coloring for computing derivatives. SIAM Review*, 47(4):629–705, 2005.

[11] J. R. Gilbert, C. Moler, and R. Schreiber. Sparse matrices in matlab: Design and implementation. *SIAM Journal on Matrix Analysis and Applications*, 13(1):333–356, 1992.

[12] G. H. Golub and C. F. Van Loan. *Matrix Computations*. Johns Hopkins University Press, Baltimore, 1996.

[13] Mini Goyal. Graph coloring in sparse derivative matrix computation. *Master's Thesis, University of Lethbridge*, 2005.

[14] Frank Harary. *Graph Theory*. Westview Press, 1994.

[15] Mahmudul Hasan. Dsjm : a software toolkit for direct determination of sparse jacobian matrices. *Master's Thesis, University of Lethbridge*, 2011.

[16] Mahmudul Hasan, Shahadat Hossain, Ahamad Imtiaz Khan, Nasrin Hakim Mithila, and Ashraful Huq Suny. Dsjm: A software toolkit for direct determination of sparse jacobian matrices. In Gert-Martin Greuel, Thorsten Koch, Peter Paule, and Andrew Sommese, editors, *Mathematical Software – ICMS 2016*, pages 275–283, Cham, 2016. Springer International Publishing.

[17] A. K. M. Shahadat Hossain and Tront Steihaug. Computing a sparse jacobian matrix by rows and columns. *Optimization Methods and Software*, 10(1):33–48, 1998.

[18] S. Hossain and T. Steihaug. Graph models and their efficient implementation for sparse jacobian matrix determination. *Discrete Applied Mathematics*, 161(12):1747–1754, 2013.

[19] S. Hossain and T. Steihaug. Optimal direct determination of sparse jacobian matrices. *Optimization Methods and Software*, 28(6):1218–1232, 2013.

[20] David Juedes and Jeffrey Jones. Coloring jacobians revisited: a new algorithm for star and acyclic bicoloring. *Optimization Methods and Software*, DOI: 10.1080/10556788.2011.606575, 2011.

[21] David W. Juedes and Jeffrey S. Jones. A generic framework for approximation analysis of greedy algorithms for star bicoloring. *Optimization Methods and Software*, DOI: 10.1080/10556788.2019.1649671, 2019.

[22] Ahamad Imtiaz Khan. Improved implementation of some coloring algorithms for the determination of large and sparse jacobian matrices. *Master's Thesis, University of Lethbridge*, 2017.

[23] Anik Saha. Bi-directional determination of sparse jacobian matrices: Algorithms and lower bounds. *Master's Thesis, University of Lethbridge*, 2015.

[24] Ashraful Huq Suny. Coloring and degeneracy for determining very large and sparse derivative matrices. *Master's Thesis, University of Lethbridge*, 2016.