IMPROVED IMPLEMENTATION OF SOME COLORING ALGORITHMS FOR THE DETERMINATION OF LARGE AND SPARSE JACOBIAN MATRICES

AHAMAD IMTIAZ KHAN
Bachelor of Science, Military Institute of Science and Technology, 2010

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

MASTER OF SCIENCE

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

IMPROVED IMPLEMENTATION OF SOME COLORING ALGORITHMS FOR THE DETERMINATION OF LARGE AND SPARSE JACOBIAN MATRICES

AHAMAD IMTIAZ KHAN

Date of Defence: August 17, 2017

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

Dedication

To my parents.

Thank you so much for offering unconditional love and support, always.

Abstract

When we solve a system of nonlinear equations or nonlinear least-squares problem by Newton's method or one of its many variants, the most computationally expensive operations per iteration are the evaluation of the Jacobian and solving the associated linear system. Many real-life problems are sparse and if we know the sparsity structure of the Jacobian in advance, great computational saving can be achieved. We revisit heuristic algorithms and sparse data structures used to determine sparse Jacobian matrices [20]. We provide a new implementation of data structures and heuristics and analyze the performance of our implementation. We provide experimental evidence of the superiority of our bucket heap data structure in terms of locality of reference to data access. Additionally, an efficient implementation of a branch-and-bound type exact coloring algorithm with new tie-breaking strategies is provided. The results are supported by extensive numerical experiments with benchmarking instances from the literature.

Acknowledgments

I am lucky that I have worked under the supervision of Dr. Shahadat Hossain. The way he treated me, it felt like he is not my supervisor instead my guardian. It may be difficult for me to work under any supervisor in future after working with such a great person. Thank you, Sir, for everything.

I want to express my sincere gratitude to my supervisory committee members, Dr. Daya Gaur and Dr. Robert Benkoczi. Their guidance, encouragement, and suggestions helped me a lot. Their immense efforts and the way of directing the students of optimization research group can be a model to others.

Without the encouragement I got from my families it would not be possible for me to come to this far and go forward. I am very grateful to my parents as well as to my parents-in-law and all the members of my two families.

I want to thank my dearest friend and my wife Lazima. We have been doing literally everything together for the last couple of years. We have been studying together, we got happy and sad together. Without her, my life would be a lot more difficult.

Nabi, Jeeshan, and Tasnuba you have made our life here happy. The support we get from you is ineffable. Thank you so much.

I also want to thank all my friends and well-wishers.

Contents

List of Tables

List of Figures

List of Symbols

The symbols given bellow are used in this thesis.

A, B, S	Uppercase letters denote matrices.
$A^\top$	Transpose of matrix A.
$a_{ij}, A(i,j)$	The entry of matrix A that is in row i and column j.
$A(i,:)$	i^{th} row of matrix A.
$A(:,j)$	j^{th} column of matrix A.
$A(:,j) \perp A(:,l)$	Column j and column l of matrix A are structurally orthogonal.
$A(:,j) \not\perp A(:,l)$	column j and Column l of matrix A are not structurally orthogonal.
$G(A)$	Column intersection graph of matrix A.
$\chi(G)$	Chromatic number of graph G.
$\mathcal{S}(A)$	Sparsity pattern of matrix A.
$\mathcal{N}_{\mathcal{J}}(j)$	Neighbors of column j of in the submatrix induced by set of columns $\mathcal{J}$.
$d_{\mathcal{J}}(j)$	Degree of column j in the submatrix induced by set of columns $\mathcal{J}$.
$\Delta_{\mathcal{J}}$	Maximum degree in the submatrix induced by set of columns $\mathcal{J}$.

$\delta_{\mathcal{J}}$	Minimum degree in the submatrix induced by set of columns $\mathcal{J}$.
m, n	Number of rows and columns of a matrix.
nnz	Number of nonzero entries of a matrix.
ρ_i	Number of nonzero entries in row i of a matrix.
ρ_{max}	Maximum number of nonzero entries in any row of a matrix.
UB	Upper bound of exact coloring algorithm.
LB	Lower bound of exact coloring algorithm.

Chapter 1

Introduction

Mathematical derivatives are required often in simulation, problems in optimization and differential equations. Solving a system of nonlinear equations or nonlinear least-squares problem by Newton's method or one of its many variants require the evaluation of its Jacobian matrix and solving the associated linear system at each iterative step. There is an ever increasing demand for solving larger and more complex problems with the advent of faster computers and sophisticated software. Fortunately, many real-life problems are sparse or otherwise structured. Evaluating the analytic derivative of a large and complex problem by hand is complicated and highly error-prone. Finite-difference (FD) scheme is an alternative to hand-coded derivatives to approximate the Jacobian. Automatic or algorithmic differentiation (AD) can be a method of choice. Using this method derivative quantities can be evaluated with an accuracy up to machine precision. It does not incur truncation error. In both methods, great computational saving can be achieved if the sparsity structure of the Jacobian is known a priori or can be computed easily and does not change from iteration to iteration.

There are two main ways one can benefit from sparsity in evaluating derivatives. First, if we know the identically zero entries of the Jacobian matrix that can be vanished then we do not need store them explicitly in a data structure. Secondly, we can speed up the calculation of sparse data by avoiding operations involving known zeros. As sparse matrix algorithms are different than their dense counterpart, distinct and more complex techniques are required for sparse matrix algorithms. A number of factors like memory traffic and the

size and organization of fast cache memory, number of floating point operations, etc affect computational complexity of sparse matrix operations. Faster cache memory is not some time large enough to hold the input data entirely when we deal with large scale problems. When we access data during the execution of an algorithm, access pattern of data is important. The term "locality of reference" is significant here. It is estimated by the principle of data locality that "recently accessed data (temporal) and nearby data (spatial) are likely to be accessed in the near future" [5]. Fewer cache misses occur due to better reference locality of data.

The purpose of this thesis is to extend the software tool DSJM (Determine Sparse Jacobian Matrices) [20] and new functionality that can be used to compress and determine sparse Jacobian matrices from its sparsity pattern using FD or AD. The current implementation provides a collection of stand-alone column ordering and grouping algorithms. This design exploits the recently proposed unifying framework "pattern graph" [20] and employs cache-friendly array-based sparse data structure. We discuss the motivation of our work and provide the problem background in the next chapter. It is easier to understand about the motivation when we talk about the problem background a little bit in detail.

1.1 Our Contributions

Our contributions in this thesis are pointed below.

1. Thorough study on the data structures used in DSJM and existing heuristic ordering and partitioning algorithms of DSJM.

2. Implementation of ordering and partitioning algorithms of DSJM using Fibonacci heap data structure and performance comparison with bucket heap data structure.

3. Cache analysis of bucket heap and Fibonacci heap-based implementations on two test instances with the help of our cache simulator.

4. Performance profiling for DSJM and ColPack[17].

5. Implementation of a branch-and-bound type exact graph coloring algorithm using efficient data structures used in DSJM.

6. Implementation of existing and new tie-breaking strategies in exact graph coloring algorithm.

7. Numerical experiments to show the efficiency of the new implementation.

8. Incorporation of our exact graph coloring implementation to the Combined coloring Method (CM) of Hossain et al. [24].

Parts of the work of this thesis,

- has appeared in ICMS: International Congress on Mathematical Software 2016 and appeared in Mathematical Software-ICMS 2016, Springer International Publishing [20].

- will be presented in a session on topic "Recent Progress in Numerical Methods and Scientific Computing" in AMMCS-2017 International Conference: Applied Mathematics, Modeling and Computational Science Conference, Waterloo, Ontario, from August 20-25, 2017.

1.2 Thesis Organization

There are a total of 5 chapters in this thesis. Chapter 1 is the introductory chapter where we introduce the problem and significance of solving the problem in general. We also discuss our contribution and thesis organization in this chapter.

Chapter 2 includes problem background and preliminaries of this thesis. We start this chapter with some definitions then discuss the direct determination problem and discuss how matrix partitioning facilitates in evaluating large and sparse Jacobian. Seed matrix calculation is essential in matrix partitioning and as seed matrix formulation is a graph

coloring problem so we discuss some graph theoretic concepts necessary for understanding the thesis afterward.

In Chapter 3, we discuss the heuristic ordering and partitioning algorithms of software toolkit DSJM. A thorough study of the algorithms is done in this chapter. Description of data structures used in DSJM is given Especially a detailed study is done on bucket data structure used in DSJM. To help understand the main idea we show the uses of bucket data structure and demonstrate how the buckets change in every step of the ordering and partitioning algorithms. We implement the algorithms of DSJM using Fibonacci heap data structure keeping all other things same. A comparison is done between Fibonacci heap-based implementation and the existing implementation. We also compare the cache misses of these two implementations during operations. Finally, we do performance profiling of DSJM and ColPack.

Chapter 4 includes a brief discussion on an exact branch-and-bound type graph coloring algorithm followed by a detailed description of the implementation of the algorithm using efficient data structures. Using a small test instance we give an example, how the branch-and-bound type exact graph coloring algorithm works. We implement four tie-breaking mechanisms for column selection which is a critical step in the algorithm. We do some numerical experiments on our implementations. We compare our implementation with Trick's [27] implementation. A comparison between the tie-breaking strategies is also given. Finally, we incorporate our exact graph coloring implementation to the Combined coloring Method (CM) of Hossain et al.

We give the concluding remarks and future work directions in the final Chapter 5

Chapter 2

Background and Preliminaries

We discuss the problem background and preliminaries necessary for this thesis in this chapter. Our problem is the determination of sparse Jacobian matrices. To become familiar with the problem we first discuss sparse matrix, Jacobian matrix then for describing the problem background we discuss determination of sparse Jacobian matrix. Then we explain matrix partitioning and why we partition columns of a matrix for determination of sparse Jacobian matrix. Finally, some graph related concept are given because matrix partitioning can be formulated as vertex coloring problem.

2.1 Sparse Matrix

A matrix is called sparse matrix if we can take computational advantages to the knowledge of its many zero entries. Structure plot of a sparse matrix is given in Figure 2.1. The sparse matrix shown in the figure is a small test matrix with 39 rows, 39 columns, and 85 non-zero entries. The matrix has total $39 \times 39 = 1521$ entries. 94% of the entries of this matrix are zeros. We say that sparsity of the matrix is 94%

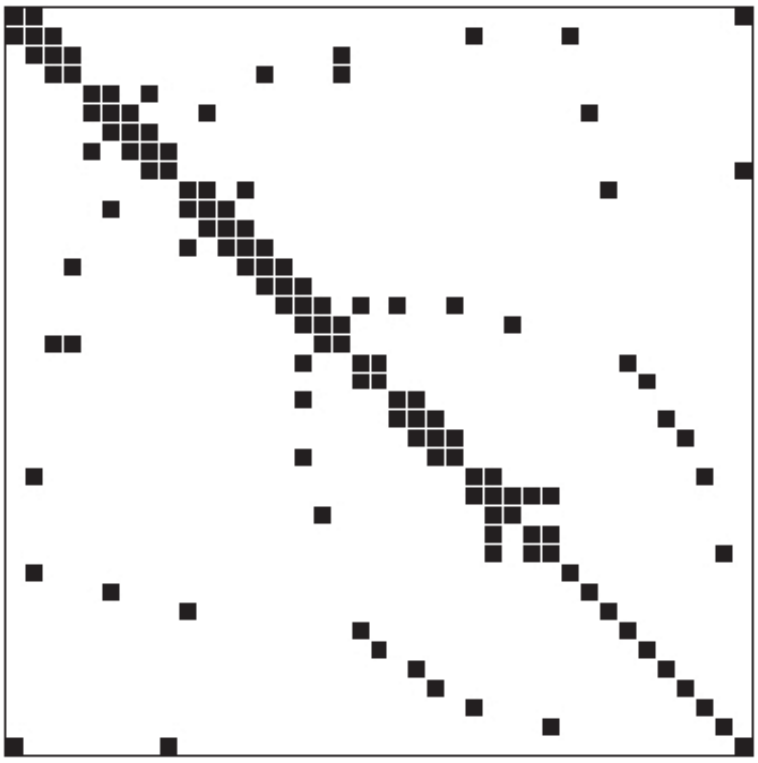

Figure 2.1: Structure plot of sparse matrix *bcspwr*01 (Dimensions: 39 $\times$ 39, 85 non-zero entires). Source [2]

2.2 Jacobian Matrix

The Jacobian matrix of a vector-valued function is the matrix of first-order partial derivatives of the vector valued function. Suppose we are given continuously differentiable mapping of a vector-valued function $F : \mathbb{R}^n \to \mathbb{R}^m$. Input of the function is vector $x \in \mathbb{R}^n$ and output vector is $F(x) \in \mathbb{R}^m$. Let $F = (f_1, f_2, \cdots, f_m)^T$ then Jacobian J of F is given by

$$J = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \frac{\partial f_m}{\partial x_2} & \cdots & \frac{\partial f_m}{\partial x_n} \end{pmatrix} \tag{2.1}$$

We discuss in the next section why it is important to calculate Jacobian of vector-valued non linear function.

2.3 Direct Determination

For solving problems in nonlinear optimization and differential equations using numerical methods, evaluation of mathematical derivatives are often required. We consider the problem of determining the Jacobian matrix $F'(x)$ of a once continuously differentiable mapping $F : \mathbb{R}^n \to \mathbb{R}^m$ at a given point $x \in \mathbb{R}^n$. Using differences, the product of the Jacobian matrix with a vector s may be approximated as

$$\left.\frac{\partial F(x+ts)}{\partial t}\right|_{t=0} = F'(x)s \approx As = \frac{1}{\varepsilon}[F(x+\varepsilon s) - F(x)] \equiv b \tag{2.2}$$

with one extra function evaluation of F at $(x+\varepsilon s)$. Here we assume that $F(x)$ has already been computed and $\varepsilon > 0$ is a small increment. Algorithmic (or automatic) Differentiation (AD) [23] forward mode gives $b = F'(x)s$ accurate up to the machine round-off, at a cost which is a small multiple of the cost of one function evaluation. The Jacobian matrix determination problem (JMDP) can be stated based on matrix-vector products as follows

> Obtain vectors $s_j \in \mathbb{R}^n, j = 1, \cdots, p$ such that the product $b_j = As_j, j = 1, \cdots, p$ or $B = AS$ determine the matrix A uniquely.

2.4 Matrix Partitioning and Seed Matrix Computation

Suppose we do not have any sparsity information. Let be s the Cartesian basis vectors $e_i, i = 1, \cdots, n$ in (2.1). Then we can obtain A with n products As. Here we need extra n function evaluations or in case of AD, n forward mode calculations.

Columns j and l of matrix A are structurally orthogonal, i.e., no two rows have non-zero entries in the same row position which can be written as $A(:, j) \perp A(:, l)$ if there does not exist any index i such that $a_{ij} \neq 0$ and $a_{il} \neq 0$. If they are not structurally orthogonal, they

are written as $A(:,j) \not\perp A(:,l)$. If $A(:,j) \perp A(:,l)$ then we need only one extra function evaluation

$$F'_j + F'_l = A(:,j) + A(:,l) = \frac{1}{\varepsilon}[F(x + \varepsilon(e_j + e_l)) - F(x)] \tag{2.3}$$

So the non-zero unknowns in columns j and l are determined from product $b = As,\ s = e_j + e_l$ directly. Curtis, Powell, and Reid [7] noted that sparsity of the Jacobian matrix can be exploited by partitioning the columns into structurally orthogonal column groups. We call this the CPR method. Thus, if the columns are partitioned into p structurally orthogonal groups, then Jacobian matrix is directly determined from the row-compressed matrix $B = AS$

So our goal is, from a known sparsity pattern of a sparse matrix, to partition its columns into the smallest number of structurally orthogonal groups such that the non-zero entries can be determined directly. One significant step in the above $B = AS$ calculation is seed matrix S calculation. We can partition the columns of the matrix as well as rows of the matrix. In the illustration below we consider column partition and how to formulate seed matrix from structurally orthogonal column groups. Let us consider the Matrix in Figure 2.2

$$A = \begin{pmatrix} 0 & a_{12} & a_{13} & a_{14} & 0 & 0 & 0 \\ 0 & a_{22} & 0 & 0 & a_{25} & 0 & 0 \\ 0 & 0 & a_{33} & 0 & 0 & a_{36} & 0 \\ 0 & 0 & 0 & a_{44} & 0 & 0 & a_{47} \\ a_{51} & 0 & 0 & 0 & a_{55} & 0 & 0 \\ a_{61} & 0 & 0 & 0 & 0 & a_{66} & 0 \\ a_{71} & 0 & 0 & 0 & 0 & 0 & a_{77} \end{pmatrix}$$

Figure 2.2: Matrix A

An approximate seed matrix is

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

We do not discuss here how we got the partitions and formulated the seed matrix S. Discussions about partitioning are given in details in the next chapters. We only point out that, each column of seed matrix S corresponds to a group of structurally orthogonal columns. The product of the sparse matrix and seed matrix is AS

$$AS = \begin{pmatrix} a_{12} & a_{13} & a_{14} \\ a_{22} & 0 & a_{25} \\ 0 & a_{33} & a_{36} \\ a_{47} & 0 & a_{44} \\ 0 & a_{51} & a_{55} \\ 0 & a_{61} & a_{66} \\ a_{77} & a_{71} & 0 \end{pmatrix}$$

Using Forward Difference (FD) the non-zero entries in columns 2 and 7 can be approximated as

$$\frac{1}{\varepsilon}(F(x+\varepsilon(e_2+e_7))-F(x))^T \approx (\tilde{a}_{12}\ \tilde{a}_{22}\ \tilde{a}_{47}\ \tilde{a}_{77})$$

Here $\varepsilon > 0$ is a small increment (step-size) and $(\tilde{.})$ indicates that the non-zero entry is an

approximation to the true value. Matrix A can be approximated using three extra function evaluation of form $F(x+\varepsilon s)$ by setting s to e_2+e_7, e_1+e_3, and $e_4+e_5+e_6$ in addition to evaluating F at x. But if we did not partition the columns we would need seven extra function evaluation of form $F(x+\varepsilon s)$ for each column in addition to evaluating F at x. In this example we use a very small test matrix having only seven columns and showed that we could directly determine Jacobian of matrix A using three extra function evaluation instead of seven. Matrix partitioning and seed matrix computation reduce the number of function evaluations. Our goal is now to reduce the number of structurally orthogonal groups so that we can efficiently determine sparse Jacobian matrices. Coleman and Moré [6] showed that the problem of finding minimum column partitioning consistent with direct determination is equivalent to a vertex coloring problem of an associated graph and that the problem is NP-Hard. So the seed matrix computation of an associated graph $G(A)$ is a vertex coloring problem. In the next section, we will discuss some necessary graph concepts.

2.5 Graph Concepts

We can represent a graph G with pair (V,E) where V is the set of vertices and E is the set of edges of graph G. Suppose we have two vertices u,v such that $u,v \in V$. These two vertices are adjacent if and only if $\{u,v\} \in E$. The neighborhood $N(v)$ is the set of all neighbors u of vertex v such that $u \neq v$ and $\{u,v\} \in E$. The degree of a vertex can be defined as $d(v) = |N(v)|$.

2.5.1 Column Intersection Graph

We can associate a graph with a matrix. Suppose we have matrix A. We can construct a graph $G = (V,E)$. In this graph G each column i of the matrix is a vertex v_i of the graph where $i = 1,2,\ldots,n$. If two columns i and j have at least one nonzero elements in the same row then those two vertices are connected with an edge i.e $\{v_i,v_j\} \in E$. Graph $G(A)$ is called the column intersection graph associated with matrix A

We display the column intersection graph $G(A)$ from matrix A of Figure 2.2 in Figure 2.3

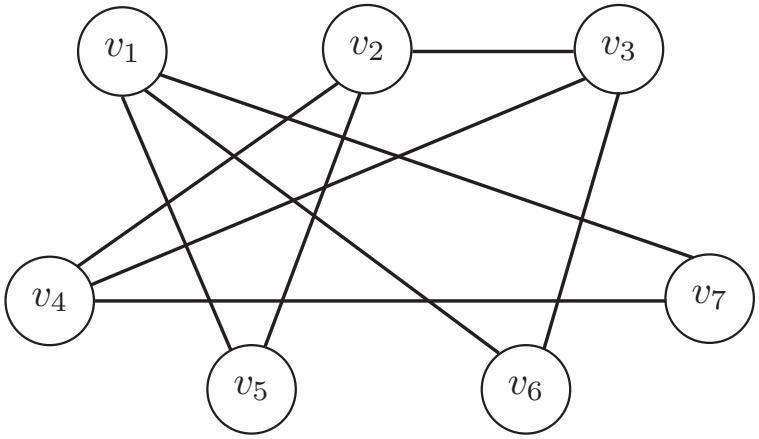

Figure 2.3: Column intersection graph $G(A)$ of matrix A

2.5.2 Graph Coloring and Coloring Methods

We have already stated that seed matrix computation in an associated graph is a vertex coloring problem. Let us discuss vertex coloring problem of a graph. The vertex coloring of a graph is assigning each vertex a color in such a way that no two adjacent vertices get the same color. Given a graph $G = (V,E)$ we call it $p-colorable$ if there is a mapping $\phi : V \rightarrow \{1,\cdots,p\}$ such that $\phi(u) \neq \phi(v)$ when $\{u,v\} \in E$. It means we have a function ϕ which assigns every vertex of graph $G(A)$ a color between 1 to p such that no adjacent vertices get the same color. The minimum value of p is denoted by chromatic number $\chi(G)$ of graph G. Given any graph G, to determine whether it is $p-colorable$ or not is NP-Complete [15].

Now we will look into graph coloring methods. In this thesis, we discuss two types of graph coloring approaches. Heuristic and exact approaches. Heuristic algorithms for graph coloring problem can be categorized into greedy constructive algorithms and meta-heuristic methods [26]. Heuristic approaches are very fast but can be very sensitive as well. The sensitivity depends on input parameter like input ordering. It has been observed that scan-

ning vertices in a specific order during coloring operation may lead to fewer colors/groups [20]. Software DSJM implements some grouping algorithms, sequential grouping (SEQ), Saturation-Degree Grouping (SD), Recursive Largest-First Grouping (RLF) and ordering algorithms Largest-First Ordering (LFO), Smallest-Last Ordering (SLO), and Incidence-Degree Ordering (IDO). Detailed Discussion about these grouping and ordering algorithms is given in Chapter 3.

Besides heuristics approaches we implement Brélaz's DSATUR based exact graph coloring algorithm [3] in DSJM. Detail discussion about exact implementation is given in Chapter 4

2.6 Forward Difference Approximation Algorithm

In Section 2.4 we have demonstrated, how non-zero entries of a column can be approximated if we have partitioning information. Suppose we have the grouping/partitioning information of structurally orthogonal groups. FD-SPJMS algorithm can be used to obtain an approximation to the nonzero entries of the directional derivative $F'(x)S(:,k) \equiv B(:,k)$ for $F \in \mathbb{R}^n \to \mathbb{R}^n$ corresponding to structurally orthogonal group k

FD-SPJMS$(gptr, gcolind, k, \eta, B)$

1 $w \leftarrow$FD-SPJD$(F, x, \eta, gptr, golind, k)$

2 **for** $ind \leftarrow gptr(k)$ **to** $gptr(k+1)-1$

3 $\quad j \leftarrow gcolind(ind)$

4 $\quad$ **for** each i for which $F'(i,j) \neq 0$

5 $\quad\quad B(i,k) \leftarrow \frac{w(i)}{\eta(j)}$

FD-SPJD is a user-defined function. It computes the difference $F(x+\eta)-F(x)$. Here η is an array which is initialized to zero and contains the finite difference increments $\varepsilon(j)$ corresponding to columns j in structurally orthogonal group k :

1 **for** $ind \leftarrow gptr(k)$ **to** $gptr(k+1)-1$

2 $\quad j \leftarrow gcolind(ind)$

3 $\quad \eta(j) \leftarrow \varepsilon(j)$

If we have the grouping information, we can efficiently find the entries of the array η to calculate the difference $F(x+\eta)-F(x)$ from which we get FD approximation in lines 2-5 of FD-SPJMS algorithm. Our concern is now how can we reduce the number of structurally orthogonal groups.

Chapter 3

Heuristic Approaches to Partitioning Algorithms

In this chapter we discuss the heuristic partitioning and ordering algorithms of DSJM. In DSJM [20] there are two kinds of partitioning algorithms. In the first kind, the columns are scanned in Smallest-Last, Largest-First or Incidence-Degree order and then colored sequentially with minimum available color. The second kind dynamically partitions the columns, i.e., column ordering and partitioning is done simultaneously. They are Saturation-Degree partitioning [3] and Recursive Largest-First partitioning [25].

In this chapter, we describe the data structures needed for the implementation first, then we discuss the algorithms and give some examples to show step by step how the algorithms work. The examples of the ordering and partitioning algorithms are given using matrix A of Figure 2.2 which has 7 rows, 7 columns, and 15 non-zero entries. Here we introduce some basic notations that are useful to understand the algorithms we discuss in this chapter.
The sparsity pattern of a matrix $A \in \mathbb{R}^{m \times n}$ is denoted

- $\mathcal{S}(A) = \{(i, j) | a_{ij} \neq 0\}$

The sparsity pattern $\mathcal{S}(A)$ of matrix A is one of the inputs in all ordering and partitioning algorithms of DSJM. Here a_{ij} is an entry of the matrix which is in row i and column j. Golub and Van Loan's [19] colon notation is used to denote submatrices. Column j of matrix a is denoted by $A(:, j)$, similarly row i is denoted by $A(i, :)$. If $\mathcal{J} = \{1, 2, \cdots, n\}$ is the set of columns of matrix A, then neighbors of column j in the submatrix induced by columns of $\mathcal{J}$, can be defined as

- $\mathcal{N}_{\mathcal{J}}(A(:,j)) = \{A(:,l), l \in \mathcal{J} | l \neq j, A(:,j) \not\perp A(:,l)\}$

In other words, neighbors of column j are all the columns $l \in \mathcal{J}$ which are structurally dependent on j. The degree of column j is the number of neighbors of $A(:,j)$ which is denoted by

- $d_{\mathcal{J}}(A(:,j)) = |\mathcal{N}_{\mathcal{J}}(A(:,j))|$

Simplified notations like $\mathcal{N}_{\mathcal{J}}(j)$ or $d_{\mathcal{J}}(j)$ are used here. The maximum and minimum degree are defined as

- $\Delta_{\mathcal{J}} = \max\{d_{\mathcal{J}}(j) | j \in \mathcal{J}\}$, $\delta_{\mathcal{J}} = \min\{d_{\mathcal{J}}(j) | j \in \mathcal{J}\}$

3.1 Data Structures

Let us discuss the data structures used to store the sparse matrices. We do not need to store all the values of a sparse matrix. Only non-zero entries are necessary for our calculations. We have to store the non-zero entries in such a way such that it exploits the sparsity as well as it is possible to associate the graph from the data structure.

3.1.1 Compressed Sparse Row(CSR) Data Structure

Compressed Sparse Row (CSR) is a popular data structure through which we can store the sparse row vectors contiguously. We can implement CSR using three arrays: *rowptr*, *colind* and *value*. In *colind* array we store the column index of each non-zero element and value of that element is stored in *value* array. *rowptr* points the column indices. *value* is a double array of size *nnz*, where *nnz* is the number of non-zero entries of the matrix. *colind* and *rowptr* are integer arrays. The size of *colind* is *nnz* and the size of *rowptr* is $m+1$, where m indicates the number of rows of the matrix. $rowptr(i)$ is the column index of first non-zero entry of row i. Suppose $rowptr(1)$ is 1. So its column index is $colind(rowptr(1))$. We can also access all non-zero entries of a row using CSR data structure. If $rowptr(1)$ is 1 and $rowptr(2)$ is 4 then we have $rowptr(2) - rowptr(1) = 4 - 1 = 3$ non-zero entries in

row 1. We get the column indices and values of the non-zero entries of row 1 form *colind* and *value* arrays. We can access elements of row i as

$$colind(rowptr(i)) \textbf{ to } colind(rowptr(i+1)-1)$$

If our matrix has m rows and nnz non-zero entries then for CSR we need $2nnz+m+1$ memory locations.

3.1.2 Compressed Sparse Column(CSC) Data Structure

We also use Compressed Sparse Column (CSC) data structure. It is similar to CSR but the three arrays are $colptr$, *rowind* and *value*. $colptr$ points the row indices. $colptr(j)$ is the row index of first nonzero element of column j. *rowind* and $colptr$ are integer arrays. The size of *rowind* is nnz and the size of $colptr$ is $n+1$, where n indicates the number of columns of the matrix. We can access elements of col j as

$$rowind(colptr(j)) \textbf{ to } rowind(colptr(j+1)-1)$$

Sparse matrices in Harwell-Boeing collection [12] are given in CSC. CSparse [9] and the MATLAB® computing environment [18] use CSC representation for sparse matrices and the associated operations.

If our matrix has n columns and nnz non-zero entries then for CSC we need $nnz+n+1$ memory locations.

In our implementation, we use both CSR and CSC data structures. So in total, we need $3nnz+m+n+2$ memory locations.

3.1.3 An Example How CSR and CSC are Used to Find the Neighbors of a Column

We already mentioned how a column intersection graph is associated with a matrix. Using CSR and CSC we can easily perform graph operations without explicitly constructing the graph. Using CSC we get the non-zero entry of any specific column and from their row index and CSR we can find if there are any non-zero entries in that row which are

the neighbors of that columns. The set of neighbors l of any column j is computed as $l = colind(indj)$ where $indj = rowptr(i) \ : \ rowptr(i+1)-1, \ i = rowind(indi), \ indi = colptr(j) \ : \ colptr(j+1)-1$. A simple and a little more elaborated example will make it easier to understand. Below, the non-zero entries of matrix A in Figure 2.2 is given then *colind*, *rowptr* arrays of CSR and *rowind*, *colptr* arrays of CSC are given in Figure 3.1 and in Figure 3.2.

In *value* array the entries are shown in row majored order. In Figure 3.1 the first three

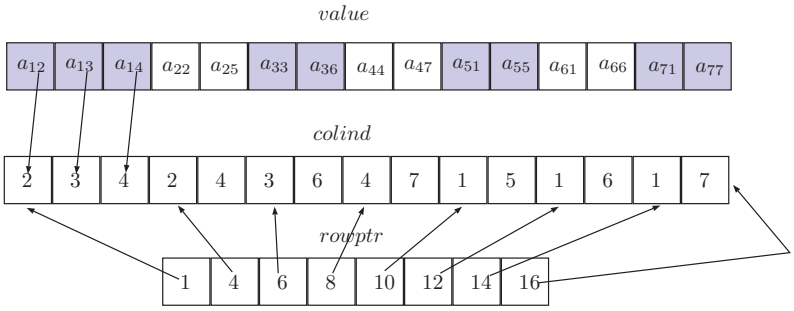

Figure 3.1: CSR data structure of matrix A

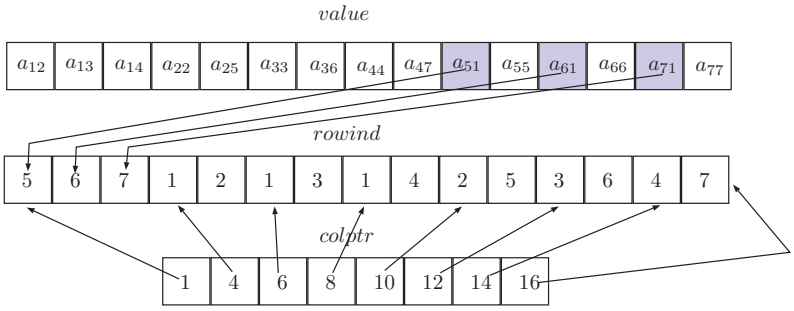

Figure 3.2: CSC data structure of matrix A

entries with gray boxes are the non-zero entries of first row, the two entries with white colored box are of second row and so on. In Figure 3.2 only entries of first column are highlighted with gray box. Let us find $l \in \mathcal{N}_\mathcal{J}(3)$ i.e neighbors of column 3 using CSR and CSC data structures. First we will compute $rowind(indi)$ from CSC. $rowind(indi)$ is the set of row indices of non-zero entries of column 3. Here $indi = colptr(j) \ : \ colptr(j+1)-1 = colptr(3) \ : \ colptr(3+1)-1 = 6:7$. So $rowind(6:7) = \{1,3\}$. Now we know the row indices of non-zero entries of column 3. So if we can find the other nonzero entries in

these two rows then we will be able to find the column indices of those rows which are the neighbors of column 3. With the help of CSR data structure we will find that. The set of neighbors is $l = colind(indj)$, For $i = 1$, $indj = rowptr(i)$: $rowptr(i+1)-1 = rowptr(1)$: $rowptr(1+1)-1 = 1:3$. Here $colind(1:3) = \{2,3,4\}$. So column 2 and 4 are neighbors of column 3. Again for $i = 3$, $indj = rowptr(i)$: $rowptr(i+1)-1 = rowptr(3)$: $rowptr(3+1)-1 = 6:7$. Here $colind(6:7) = \{3,6\}$. so column 6 is another neighbor of column 3. These CSR and CSC are the backbones of our computation. We do not need to construct a graph explicitly. We can compute the degree of each column because we know how to access the neighbors of each column using CSC and CSR. The matrix elements of CSR and CSC data structure are placed in contiguous locations in the computer memory which ensures maximum spatial locality [29] which results in better cache memory utilization.

3.1.4 Bucket Data Structure and Tagging Scheme

Besides CSR and CSC, most frequently used sparse matrix operations in the implementation of algorithms are

1. an efficient tagging scheme for tagging processed and unprocessed columns.

2. a bucket data structure [14] to efficiently find a column with the minimal/maximal degree.

The bucket data structure we use for implementing the algorithms is very efficient. With the help of this data structure, we build min/max priority queues. We call those bucket heaps. This data structure stores and holds the columns based on their degrees in some induced subgraph. The columns of the same degree are stored in same degree list. When the algorithms process the columns, the degrees of the columns may change, in those cases, we update the degrees of those columns and change their degree lists. When we implement bucket data structure, we do not use pointers. Instead, we use three arrays named $HEAD$, $PREVIOUS$ and $NEXT$. The size of $HEAD$ is *maxdeg*, where *maxdeg* is the maximum

degree of the associated graph. The size of $PREVIOUS$ and $NEXT$ is $n+1$. So the overall storage requirement is $maxdeg+2n+2$. Array $HEAD$ holds the indices of the first column of each degree list. Suppose $HEAD(2)=4$. It means the first column of degree list 2 is column 4. The degree lists of the bucket are based on column degree, incidence degree or saturation degree. When the algorithms perform ordering or partitioning the degree/incidence degree/saturation degree of column(s), change and degree list of column(s) are updated accordingly. The array $HEAD$ keep track of the first column of each degree list. $NEXT$ and $PREVIOUS$ arrays are used to keep track of next and previous column of each column in its degree list. Suppose $NEXT(4)=6$. It means column 6 is the next column of column 4. $PREVIOUS$ array also works in the same way. If for any column j, $NEXT(j)=0$ it means it is the last column of its degree list. If for any column j $PREVIOUS(j)=0$ it means it is the first column of its degree list. Index of $HEAD$ starts from 0 but indices of $NEXT$ and $PREVIOUS$ start from 1.

Two important operations in bucket heap are ADD and $DELETE$. The purpose of these operations is to add a column or delete a column from the bucket. Suppose in a step of an ordering algorithm a column with the minimum degree is selected. The column gets deleted from the bucket and stored in the ordered list. Here $DELETE$ is used. As the column is deleted from the unordered list of columns, so the degrees of its neighbors are decreased by 1. So we delete them from their current degree list d and add them to degree list $d-1$. The ADD and $DELETE$ column operations are showed in Figure 3.3.

```
ADD(j,d)
1  PREVIOUS(j) ← 0
2  NEXT(j) ← HEAD(d)
3  if HEAD(d) > 0
4        PREVIOUS(HEAD(d)) ← j
5  HEAD(d) ← j
```

Figure 3.3: Bucket heap ADD operation

DELETE(j,d)
1 **if** $PREVIOUS(j) = 0$
2 $\quad HEAD(d) \leftarrow NEXT(j)$
3 **else**
4 $\quad NEXT(PREVIOUS(j)) \leftarrow NEXT(j)$
5 **if** $NEXT(j) > 0$
6 $\quad PREVIOUS(NEXT(j)) \leftarrow PREVIOUS(j)$

Figure 3.4: Bucket heap DELETE operation

With the help of bucket data structure, we discuss the ordering and partitioning algorithms and describe the steps of the algorithms with examples in the next section. We try to keep the examples as simple as possible. We perform the ordering and partitioning algorithms on the matrix shown in Figure 2.2

3.2 Algorithms

3.2.1 Sequential Partitioning (SEQ)

This algorithm is a variant of CPR partitioning. The columns are initially ordered in Largest-First, Smallest-Last or Incidence-Degree basis then the columns are assigned to the minimum group number. The SEQ algorithm is shown in Figure 3.5 below.

SEQ$(\mathcal{S}(A), group, ngroup, order)$
1 $ngroup \leftarrow 0$
2 $\mathcal{J} \leftarrow \{1,2,\ldots,n\}$
3 **for** $k \leftarrow 1$ **to** n
4 $\quad j \leftarrow order(k)$
5 $\quad$ let $c_m = \min\{c | c \in \{1,\ldots,ngroup+1\} \neq group(l), l \in \mathcal{N}_{\mathcal{J}}(j)\}$
6 $\quad group(j) \leftarrow c_m$
7 $\quad$ **if** $c_m > ngroup$
8 $\quad\quad ngroup \leftarrow ngroup + 1$

Figure 3.5: Sequential Partitioning algorithm

The inputs of SEQ algorithm are the sparsity pattern of matrix A, an array named *group*

with all its elements initialized to the number of columns n and an array of scanning order called *order*. In line 5 of Figure 3.5 set $\mathcal{N}_{\mathcal{J}}(j)$ represents the set of neighbors of column j and those cannot be grouped with column j. In the same line c_m represent the least-numbered structurally orthogonal group to which column j can be included. On the termination of the algorithm the group of any column j can be found in $group(j)$ and *ngroup* holds the total number of structurally orthogonal groups in the resulting partition.

In SEQ algorithm in lines 1 and 2 initialization takes constant time. It takes $O(n)$ steps. In line 5 the algorithm looks for the neighbors to find c_m, the minimum numbered available color. In our implementation to find the neighbors for each non-zero entries of the selected column we look for the non-zero entries in the same row. So in total each row is searched ρ_i^2 times where ρ_i is the number of nonzero elements in the row i. If we have m rows, sequential coloring will take $O(\sum_{i=1}^{m} \rho_i^2)$ operations

3.2.2 Smallest-Last Ordering (SLO)

Suppose we have ordered a set of vertices $V' = \{v_n, v_{n-1}, \ldots, v_{i+1}\}$ of graph $G(A)$. The algorithm chooses the i-th vertex u to place it in ordered set of vertices from unordered set of vertices such that $deg(u)$ is minimum in $G[V \backslash V']$. The SLO algorithm is shown in Figure 3.6.

```
SLO(S(A), order)
1  slindex ← n
2  J ← {1, 2, ..., n}
3  while J ≠ Ø
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: Smallest-Last Ordering algorithm

The input of the SLO algorithm is the sparsity pattern of matrix A. This algorithm gives

the array *order* when it terminates. The kth position for $k = 1, \ldots, n$ in smallest-last order is the column with index $order(k)$.

3.2.3 SLO step by step

In SLO the degree list is based on the degrees of columns in $G[V \backslash V']$ where V is the set of all vertices and V' is the set of ordered vertices. Initially V' is empty. We construct a bucket and put the columns in different degree lists based on their degrees in V. For constructing the bucket we need the degree of each column, and we have an array *ndeg* which contains the degrees of all columns. Initially, the bucket looks like as follows.

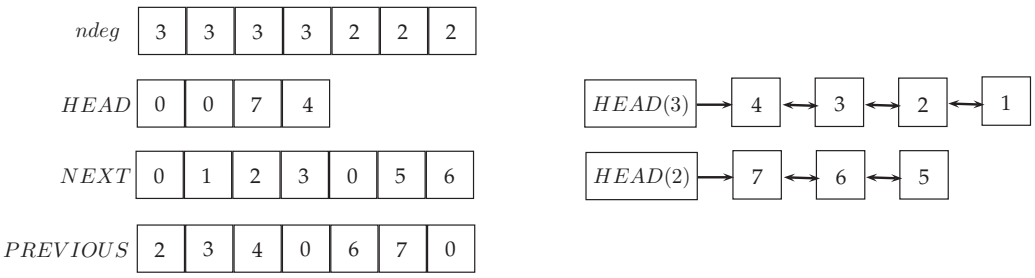

Figure 3.7: Bucket of matrix A after initialization in SLO

From the *ndeg* array we see that degree of column 1, 2, 3 and 4 is 3 and degree of column 5, 6 and 7 is 2. Initially, we construct two degree lists. The right part of Figure 1.3 shows the two lists and the columns under those lists. $HEAD(2)$ is 7. It means the first column of this degree list 2 is 7. Similarly the first column of $HEAD(3)$ is 4. $NEXT$ and $PREVIOUS$ arrays hold the next and previous columns of each column. $NEXT(3)$ is 2 and $PREVIOUS(3)$ is 4. From the degree list 3, we see that next and previous of column 3 are column 2 and 4.

Now in the first iteration, the algorithm chooses the column with the minimum degree. Line 4 of Figure 3.6 does this. It is column 7. In line 5 the algorithm puts this column in ordered list then in line 7 deletes this column from unordered list. As this column gets deleted from unordered list, degrees of all of its neighbors are also updated. The neighbors of column 7 are column 1 and 4. So their degrees are updated, and they get removed from

their current degree list and inserted into a new degree list. After the first iteration, the bucket looks like this.

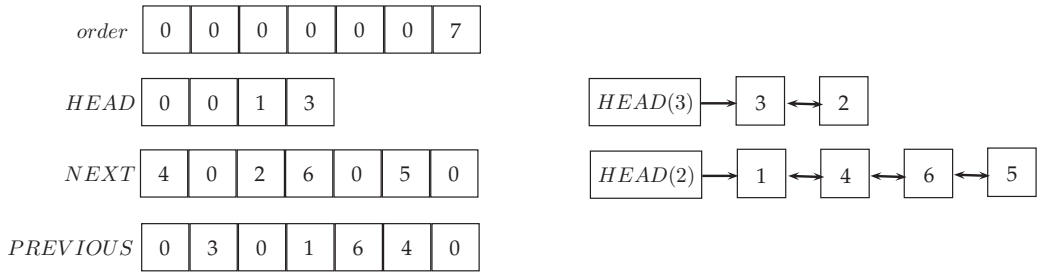

Figure 3.8: Bucket after first iteration in SLO

After the bucket construction, the *ndeg* array is not needed. So in the next iterations, we show the *order* array instead of *ndeg*. *order* array is a list that stores the columns in smallest last order.

In the next iteration, column 1 is chosen, and the bucket looks like as follows

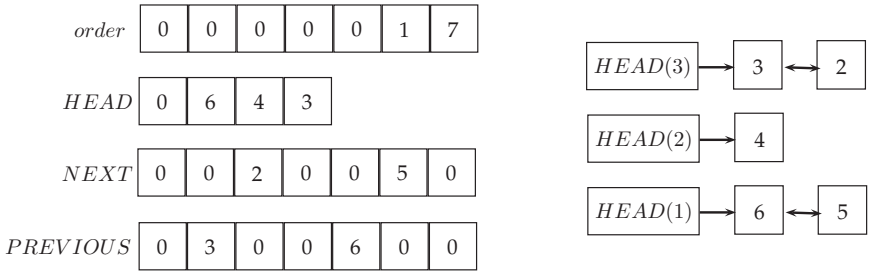

Figure 3.9: Bucket after second iteration in SLO

In the third iteration, column 6 is chosen and the bucket looks like as follows

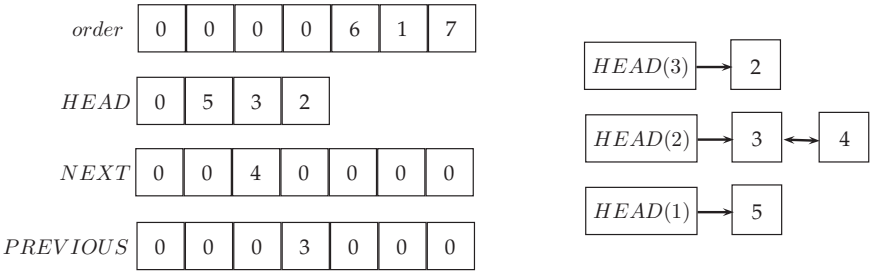

Figure 3.10: Bucket after third iteration in SLO

In the fourth iteration, column 5 is chosen and the bucket looks like as follows

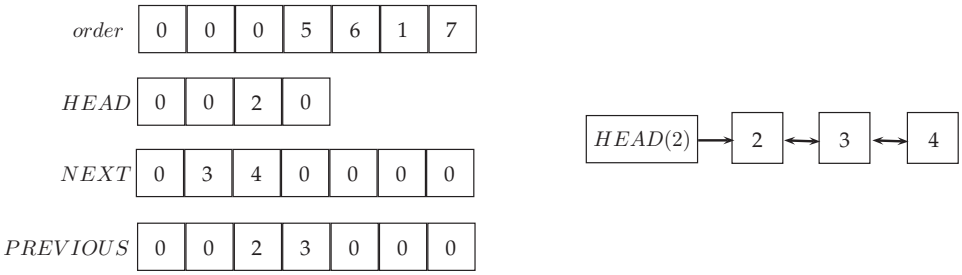

Figure 3.11: Bucket after fourth iteration in SLO

In the fifth iteration, column 2 is chosen and the bucket looks like as follows

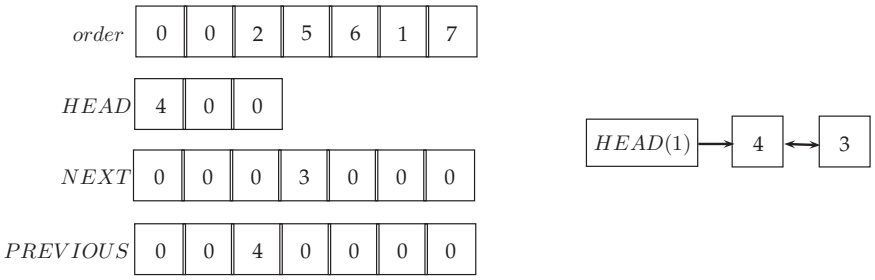

Figure 3.12: Bucket after fifth iteration in SLO

In the sixth iteration column 4 is chosen, and in the subsequent iteration, there is only one column left. That is column 3. After putting the last column in ordered list line 3 of Figure 3.6 becomes false, and the algorithm terminates. Figure of bucket after sixth iteration and the final ordered list is shown bellow

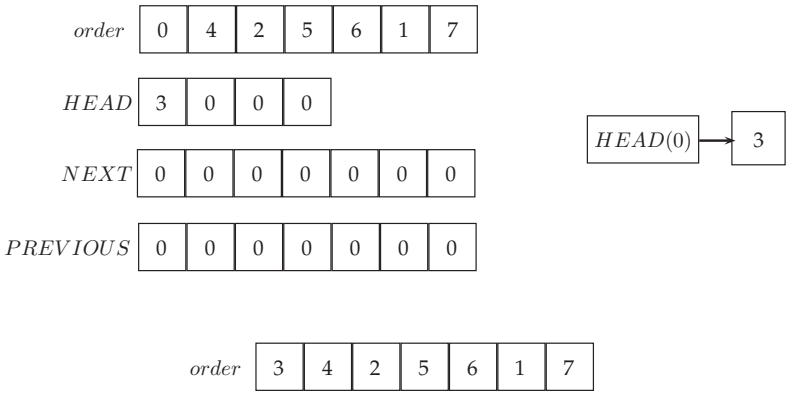

Figure 3.13: Bucket after sixth and final iterations in SLO

In SLO algorithm in lines 1 and 2 initialization takes constant time. It takes $O(n)$

steps. In line 4 the column with the minimum degree is selected. It is a constant time operation. Lines 5 and 6 are also constant time operations. In line 7 the selected column with the minimum degree gets deleted from the graph. The largest computational cost in this algorithm is to delete the column with the minimum degree. Deleting a column means the algorithm updates the degrees of all of its adjacent columns. For updating the degrees of the adjacent columns in every iteration, one nonzero element of the column is selected then the algorithm looks for every non-zero entries in the same row. So each row is searched ρ_i^2 times where ρ_i is the number of nonzero elements in the row i. If we have m rows, deleting column and updating their neighbors will take $O(\sum_{i=1}^{m} \rho_i^2)$ operations.

3.2.4 Incidence-Degree Ordering (IDO)

Suppose we have a ordered set of vertices $V' = \{v_1, v_2, \ldots, v_{i-1}\}$ of graph $G(A)$. The algorithm chooses the i-th vertex u to place it in ordered set of vertices from the unordered set of vertices such that $deg(u)$ is maximum in $G[V']$. If there are more than one vertex with the maximum degree then the vertex with the largest degree in $G[V \backslash V']$ is chosen to break the tie. The IDO algorithm is given in Figure 3.14.

IDO$(\mathcal{S}(A), order)$
1 $idindex \leftarrow 1$
2 $\mathcal{J} \leftarrow \{1, 2, \ldots, n\}$
3 $\mathcal{U} \leftarrow \mathcal{J}$
4 $\mathcal{O} \leftarrow \emptyset$
5 **while** $\mathcal{U} \neq \emptyset$
6 $\quad \mathcal{M} \leftarrow \{l \in \mathcal{U} \mid d_{\mathcal{O}}(l)$ is maximum$\}$
7 $\quad$ let $j \in \mathcal{M}$ be such that $d_{\mathcal{U}}(j) + d_{\mathcal{O}}(j)$ is maximum
8 $\quad order(idindex) \leftarrow j$
9 $\quad idindex \leftarrow idindex + 1$
10 $\quad \mathcal{O} \leftarrow \mathcal{O} \cup \{j\}$
11 $\quad \mathcal{U} \leftarrow \mathcal{U} \setminus \{j\}$

Figure 3.14: Incidence-Degree Ordering algorithm

The input of the IDO algorithm is the sparsity pattern of matrix A. This algorithm gives

the array *order* when it terminates. The kth position for $k = 1, \ldots, n$ in incidence-degree order is the column with index $order(k)$.

3.2.5 IDO step by step

In IDO the degree list is based on incidence degree of unordered columns in the subgraph induced by ordered viertices of graph $G[V']$. Here V' is the set of ordered vertices. We call this incidence degree of vertices. Initially V' is empty. Incidence degree of all unordered columns is 0, so the algorithm inserts all columns in degree list 0. For ordering, the degree of columns in induced subgraph $G[V \backslash V']$ is needed. We store degree of columns in $G[V \backslash V']$ in array *ideg*. At the beginning V' is empty so *ideg* is as same as *ndeg*. Initially, the bucket looks like as follows.

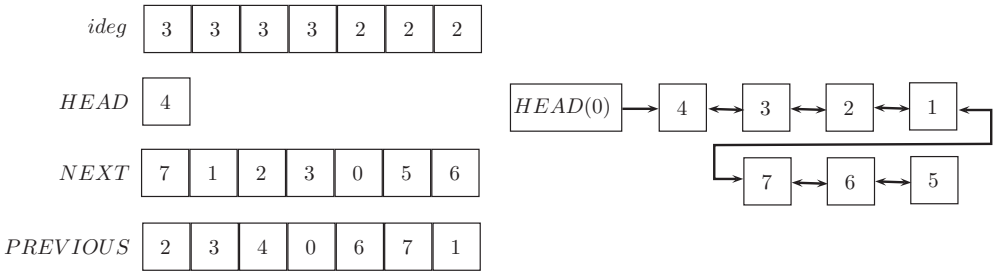

Figure 3.15: Bucket of matrix A after initialization in IDO

After initialization at line 6 of Figure 3.14, it will look for all column with highest incidence degree. At the beginning, the highest incidence degree is 0. To break the tie, it will choose column with the maximum degree in induced subgraph in line 7. Column 4 is chosen. It is then stored in the array *order*. In line 10 the selected column is added to the graph of ordered columns, and in line 11 it is removed from the graph of unordered columns. In the implementation, we do the same thing but in a little bit different way. We do not construct a graph of ordered columns explicitly rather the degree lists of the neighbors of the selected column are updated. Here column 4 is selected. Column 4 is deleted from the bucket and is stored in ordered list of columns. The neighbors of column 4 are 2, 3 and 7. Initially, they were in degree list 0. After selecting column 4, they get removed from

degree list 0 and inserted into a higher degree list 1. This is how we change the $d_O(j)$ for any column j. So now $d_O(2) = d_O(3) = d_O(7) = 1$ and $d_O(1) = d_O(5) = d_O(6) = 0$.
As we removed column 4, the degrees of its neighbors are also updated in the array *ideg*.

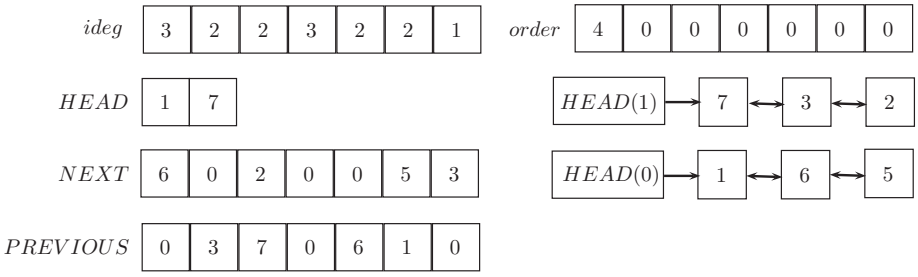

Figure 3.16: Bucket after first iteration in IDO

After the first iteration, the maximum incidence degree is 1. So all columns from degree list 1 are selected. From the selected columns, the column with the maximum degree in the graph of unordered columns is chosen which is column 3. Column 3 is added to ordered list, and incidence degrees of its neighbors are also increased. Neighbors of column 3 in the unordered graph are 2 and 6. So they get moved to higher degree lists.

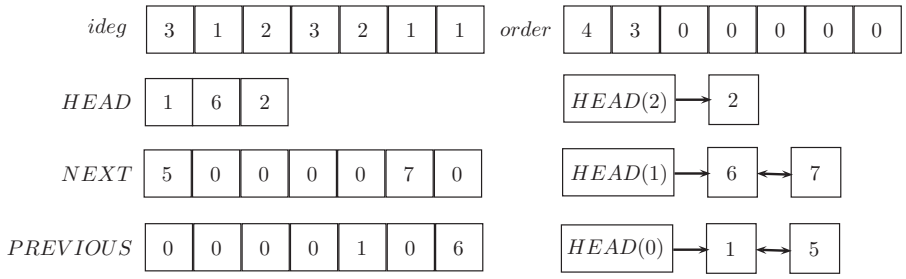

Figure 3.17: Bucket after second iteration in IDO

Maximum incidence degree is now 2, and we have only one column in this degree list. So column 2 is chosen and stored in ordered list.

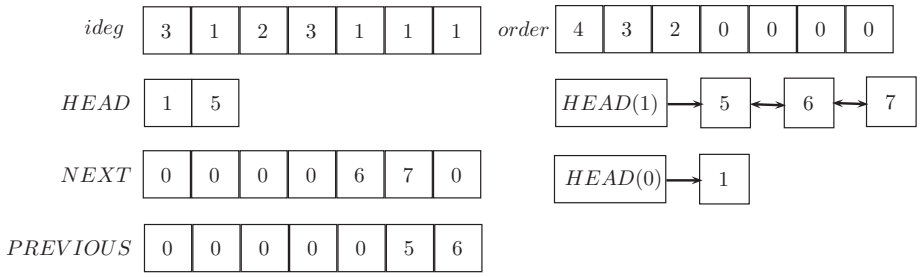

Figure 3.18: Bucket after third iteration in IDO

Now maximum incidence degree is 1 and column 5, 6 and 7 are selected. Degrees of column 5,6 and 7 in induced subgraph are also same. So the first column is chosen from degree list, i.e., column 5.

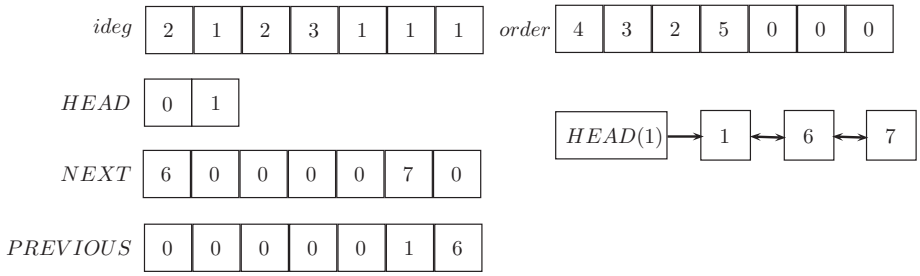

Figure 3.19: Bucket after fourth iteration in IDO

From maximum degree list column 1, 6 and 7 are selected. Column 1 has the maximum degree in the induced subgraph, so it is chosen.

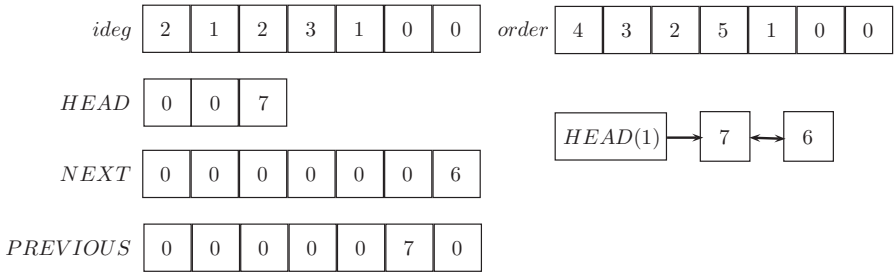

Figure 3.20: Bucket after fifth iteration in IDO

In iteration 6, column 7 is chosen from maximum degree list 2, and in the final iteration the only column left 6 is chosen and stored in *order*.

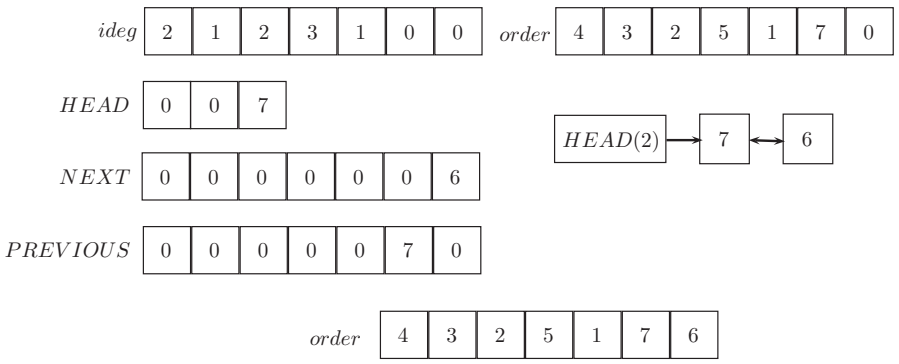

Figure 3.21: Bucket after sixth and final iterations in IDO

Like SLO line 11 of IDO algorithm involves with largest computational cost. In this step selected column gets deleted from the unordered list of vertices and incidence degrees of all of its adjacent columns are updated. The computational cost is exactly same as updating degrees of adjacent columns of selected smallest degree column in SLO. So IDO also requires $O(\sum_{i=1}^{m} \rho_i^2)$ operations.

3.2.6 Saturation-Degree Partitioning (SDPartition)

Suppose we have ordered and colored a set of vertices $V' = \{v_1, v_2, \ldots, v_{i-1}\}$ of graph $G(A)$. The algorithm chooses the i-th vertex u to place it in ordered and colored set of vertices from the unordered set of vertices such that $kdeg(u)$ is maximum in $G[V']$. Here $kdeg(u)$ stands for chromatic degree of vertex u. Chromatic degree of u is the number of different color(s) in neighborhood of u. If there are more than one vertices with the maximum degree then the vertex with the largest degree in $G[V \backslash V']$ is chosen to break the tie. The SDPartition algorithm is given in Figure 3.22

In SDPartion the column with maximum $kdeg$ is chosen. If there are two or more than such columns then column j is chosen that has the largest degree in the subgraph induced by unordered columns. From the color group C the minimum available color c_m is chosen and assigned such that j is structurally orthogonal with columns in that color group.

SDPARTITION$(\mathcal{S}(A), group, ngroup)$

```
1  ngroup ← 0
2  J ← {1,2,...,n}
3  U ← J
4  P ← Ø
5  while U ≠ Ø
6      M ← {l ∈ U | kd_P(l) is maximum}
7      let j ∈ M be such that kd_P(j) + d_U(j) is maximum
8      let c_m =min{c|c ∈ {1,...,ngroup+1} ≠ group(l), l ∈ N_J(j)}
9      group(j) ← c_m
10     if c_m > ngroup
11         ngroup ← ngroup + 1
12     P ← P ∪ {j}
13     U ← U \ {j}
```

Figure 3.22: Saturation-Degree Partitioning algorithm

3.2.7 SDPartition step by step

In SDPartition the degree list is based on chromatic degrees of uncolored columns in colored graph $G[V']$ where V' is the set of colored vertices. We call this saturation degree of vertices. Initially V' is empty. Saturation degree of all columns is 0, so the algorithm in Figure 3.22 inserts all columns in degree list 0. For choosing the column for coloring, the degree of columns in induced subgraph $G[V \backslash V']$ is needed. We store degree of columns in $G[V \backslash V']$ in array *ideg*. At the beginning V' is empty so *ideg* is as same as *ndeg*. Unlike the previous SLO and IDO algorithms SDPartition algorithm colors one column in each iteration. The colors of the columns are stored in the *color* array. Initially, the bucket looks like as follows.

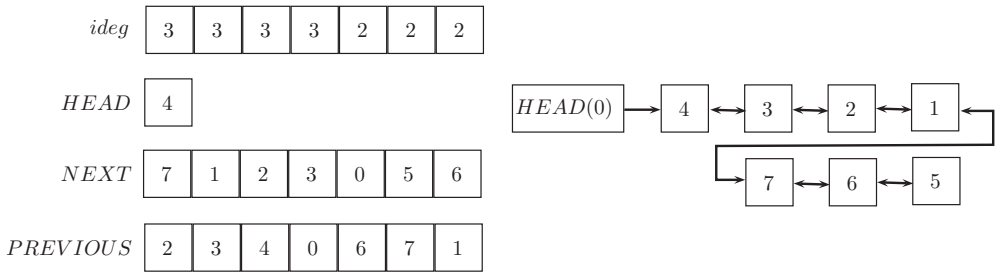

Figure 3.23: Bucket of matrix A after initialization in SDPartition

After initialization at line 5 of the SDPartition algorithm in Figure 3.22, it looks for all columns with highest saturation degree. In the beginning, the highest saturation degree is 0. To break the tie, it will choose column with the maximum degree in induced subgraph in line 6. Column 4 is chosen. The algorithm then looks for the minimum color available for column 4. The minimum available color is 1 so this color stored in the array *color* at index 4. In line 9 the selected column is added to the graph of colored columns, and in line 10 it is removed from the graph of uncolored columns. In the implementation, we do the same thing but in a little bit different way. We do not construct a graph of colored columns explicitly rather the degree lists of the neighbors of the selected column are increased. Saturation degrees of all neighbors are not increased. If the color of the selected column is new for its neighbors, then their saturation degrees are updates. In the algorithm the chromatic/saturation degree of the any column j is denoted by $kd_{\mathcal{P}}(j)$ (lines 6 and 7). At first column 4 is selected. The neighbors of column 4 are 2, 3 and 7. Initially, they are in degree list 0. After selecting column 4, it is colored using color 1 and is removed from the bucket. As none of column 2,3 and 7 have any other neighbor with color 1 in the colored graph, they get removed from degree list 0 and inserted into a higher degree list 1. This is how we change the $kd_{\mathcal{P}}(j)$ for any column j. Now $kd_{\mathcal{P}}(2) = kd_{\mathcal{P}}(3) = kd_{\mathcal{P}}(7) = 1$ and $kd_{\mathcal{P}}(1) = kd_{\mathcal{P}}(5) = kd_{\mathcal{P}}(6) = 0$.

As we remove column 4 from the bucket that means it is removed from the uncolored set of columns, the degrees in the induced subgraph of its neighbors are also updated in the array *ideg*.

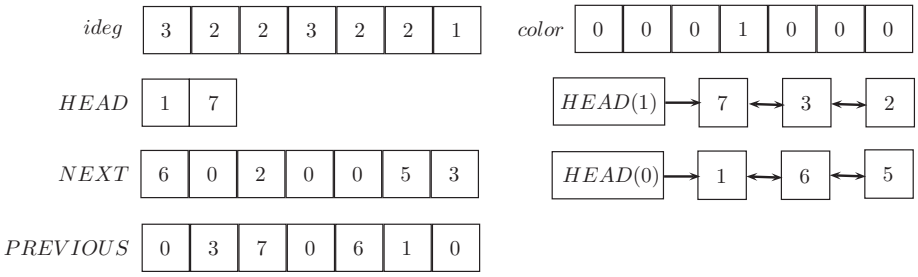

Figure 3.24: Bucket after first iteration in SDPartition

After the first iteration, the maximum saturation is 1. So all columns from degree list 1 are selected. Then the column with the maximum degree in the unordered graph is chosen that is column 3. Column 3 is then colored with color 2 as its neighbor column 4 was already colored with 1 so color 1 is not available. Saturation degrees of its neighbors are also increased. Neighbors of column 3 in the unordered graph are 2 and 6. For both column 2 and 6, a neighbor with color 2 is new. So they get moved to higher degree lists.

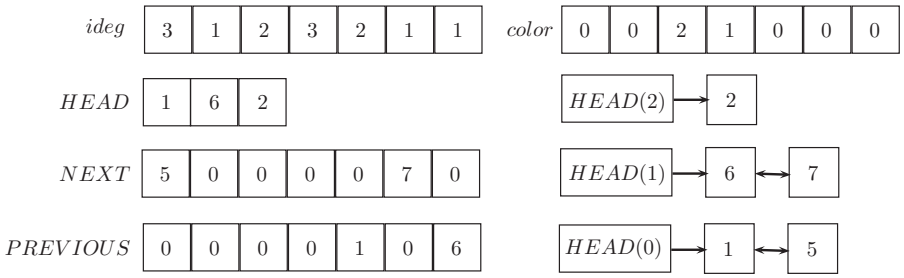

Figure 3.25: Bucket after second iteration in SDPartition

Maximum saturation degree is now 2 and we have only one column in this degree list. So column 2 is chosen and colored with minimum available color 3.

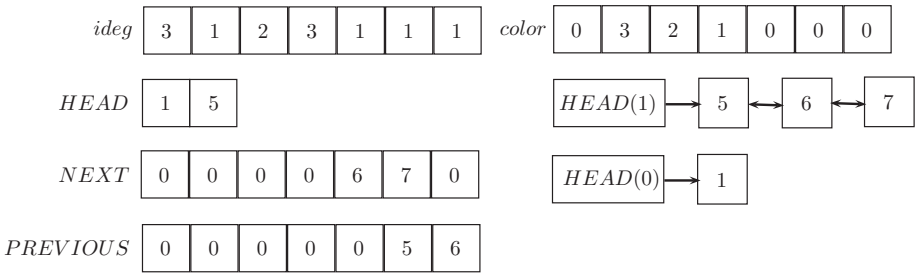

Figure 3.26: Bucket after third iteration in SDPartition

Now maximum saturation degree is 1 and column 5, 6 and 7 are selected. Their degrees in induced subgraph are also same. So the first column is chosen and colored from degree list i.e column 5.

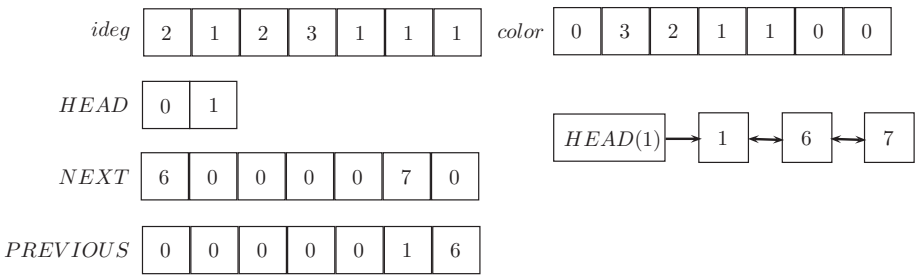

Figure 3.27: Bucket after fourth iteration in SDPartition

From maximum degree list column 1, 6 and 7 are selected. Column 1 has the maximum degree in induced subgraph so it is chosen.

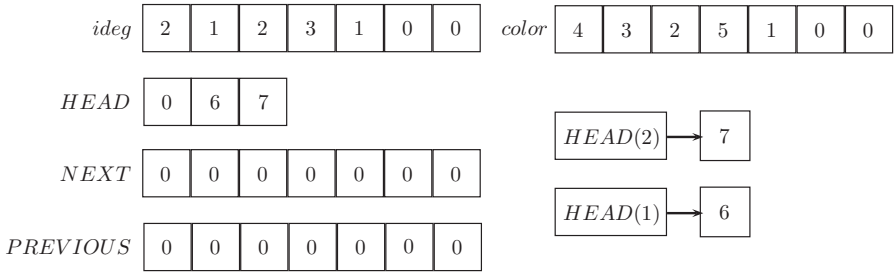

Figure 3.28: Bucket after fifth iteration in SDPartition

In iteration 6, column 7 is chosen from maximum degree list 2 and in final iteration the only column left 6 is chosen and its color is stored in *color*

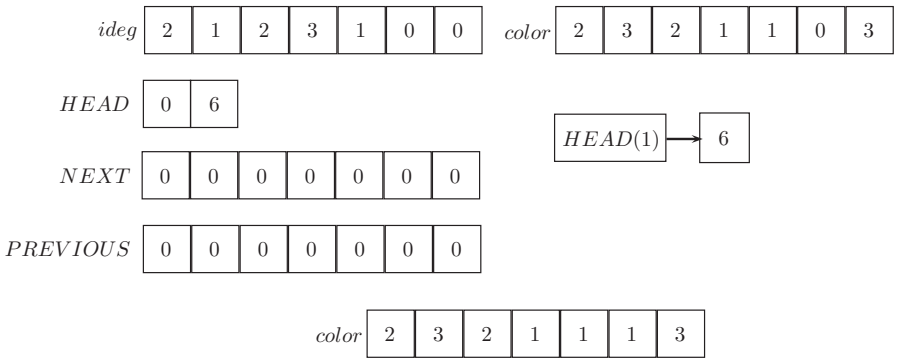

Figure 3.29: Bucket after sixth and final iterations in SDPartition

Two operations incur largest computational cost in SDPartition. In line 8 minimum available color c_m is selected. For selecting the minimum color of a column, the algorithms

looks for all its adjacent columns. So for each non-zero entries in that column, the algorithm looks for non-zero entries in each row. So to find minimum color for a column i total $\sum_{\substack{a_{ij}\neq 0 \\ i=1}}^{m} \rho_i$ operations required. To color all columns total number of required operations is $\sum_{j=1}^{n} \sum_{\substack{a_{ij}\neq 0 \\ i=1}}^{m} \rho_i = \sum_{i=1}^{m} \rho_i^2$.
Another critical operation in SDPartition is deleting the selected column and updating its adjacent columns' chromatic degrees. In SLO and IDO degrees/incidence degrees of the adjacent columns are updated, but in SDPartition, we increase saturation degrees of the adjacent columns if the color of the selected column is a new color for its neighbors in the uncolored set of columns. Previously what we did was, for selected column j we checked all $j' \in adj(j)$. For each column in j' we checked all $j'' \in adj(j')$. If any column of set j'' has color as same as the color of column j then saturation degree of j' was not increased. This operation is an expensive operation. Later we modified this operation and used bitset to perform the same thing. Now updating the saturation degree by checking bitset is a constant time operation. So deleting the selected column and updating its neighbors' saturation degrees take the same number of operations we do to delete column and update its adjacents' degrees in SLO and IDO. So overall the running time of SDPartition is $O(\sum_{i=1}^{m} \rho_i^2)$

3.2.8 Recursive Largest-First Partitioning (RLFPartition)

RLFPartition algorithm partitions the vertex set V into $V_1, V_2, \ldots, V_p$ independent sets. The independent sets are structurally orthogonal column partition, i.e., each independent set is a set of vertices in which there are no edges between the member vertices. The RLFPartition algorithm is given in Figure 3.30

Here $\mathcal{U}$ is the set of indices of column that are not grouped yet. A column is chosen $l \in \mathcal{U}$ such that $d_{\mathcal{U}}(l)$ is maximum. Then the columns in $\mathcal{U}$ are partitioned in three sets: I which represents structurally orthogonal set we are currently constructing, $\mathcal{F}$ is the set of columns that are neighbors of l which are inadmissible to I and $\mathcal{X}$ is the set of columns of $\mathcal{U}$ that are not in $\mathcal{F}$. Initially the algorithm includes l in I and neighbors of l in $\mathcal{U}$ i.e

```
RLFPARTITION(S(A), group)
 1  U ← {1,2,...,n}
 2  gnum ← 0
 3  I ← Ø
 4  while U ≠ Ø
 5      let l ∈ U be such that d_U(l) is maximum
 6      U ← U \ {l}
 7      F ← N_U(l)
 8      X ← U \ {F}
 9      I ← I ∪ {l}
10      while X ≠ Ø
11          let j ∈ X be such that d_F(j) is maximum
12          I ← I ∪ {j}
13          U ← U \ {j}
14          F ← F ∪ N_X(j)
15          X ← X \ (N_X(j) ∪ {j})
16      gnum ← gnum + 1
17      for j ∈ I
18          group(j) ← gnum
19      I ← Ø
```

Figure 3.30: Recursive Largest-First Partitioning algorithm

$\mathcal{N}_{\mathcal{U}}(l)$ are included in $\mathcal{F}$. Then the algorithm recursively selects a column j from $\mathcal{X}$ that has maximum degree in $\mathcal{F}$.

3.2.9 RLFPartition step by step

RLFPartition constructs structurally orthogonal group until all columns are included in a group. In line 1 of RLFPartition algorithm in Figure 3.30, $\mathcal{U}$ is initialized. This $\mathcal{U}$ is the set of indices of columns that are not grouped yet. Initially, all columns are in this set. Based on this set we construct our bucket *priority_queue*. Initially, we construct this *priority_queue* bucket based on the degrees of the columns. In line 5 of Figure 3.30 a column with the maximum degree is picked. The algorithm picks that column from the highest degree list of *priority_queue*. This is the first column of a structurally orthogonal column group. The algorithm will try to add as many columns as possible to this group. In lines 7 and 8,

all neighbors of selected column are chosen and put into a forbidden set $\mathcal{F}$ because all of these columns are not structurally orthogonal columns with the selected column for sure. Next target of the algorithm is to add columns in the structurally orthogonal set I. For that, a set $\mathcal{X}$ is constructed which is $\mathcal{U}\backslash\mathcal{F}$. From this set a column is chosen which has the maximum degree in forbidden set $\mathcal{F}$. To perform this operation we do not construct a bucket for the forbidden set explicitly instead we form a bucket *u_queue* that contains all the columns in $\mathcal{X}$ and their degree lists differ based on their degrees in forbidden set $\mathcal{F}$. The algorithm recursively takes columns from *u_queue* and adds them to the structurally orthogonal group until *u_queue* is empty. Initially the *priority_queue* and *u_queue* bucket looks like as follows. As there is no column in the forbidden set, so all the columns are in degree list 0 of *u_queue*. The buckets are shown in Figure 3.31

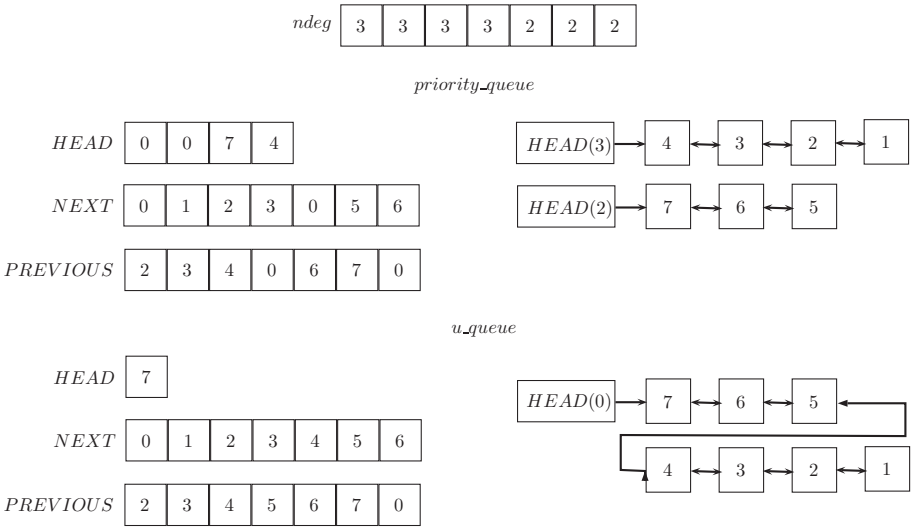

Figure 3.31: *priority_queue* and *u_queue* buckets after initialization in RLFPartition

At first column from *priority_queue* with maximum degree is chosen and added to a new structurally orthogonal group 1. The chosen column is 4. All of the neighbors of column 4 are now in forbidden set. They are column 2,3 and 7. So $\mathcal{X} = \mathcal{U}\backslash F = \{1,2,3,4,5,6,7\}\backslash\{2,3,4,7\} = \{1,5,6\}$. Here forbidden set $\mathcal{F} = \{2,3,4,7\}$. Degree of column 1, 5 and 6 in this $\mathcal{F}$ is 1 so all these columns are in degree list 1 of *u_queue*.

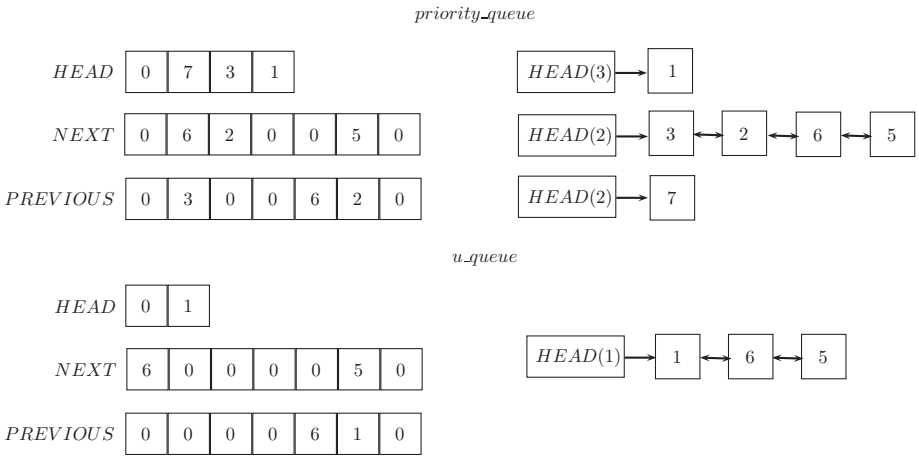

Figure 3.32: Buckets after first iteration in RLFPartition

The algorithm will recursively add columns from this *u_queue* until it is empty. As the the selected column 4 get removed from the *priority_queue* their neighbors degree lists also get changed.

u_queue is not empty. So in the next step from *u_queue* column with maximum degree will be chosen. It is column 1. After adding column 1 and its neighbors to forbidden set $\mathcal{F}$, *u_queue* becomes empty. So the algorithm is done with adding all structurally orthogonal columns to group 1. They are column 4 and 1. We color both of these columns with color 1 and store the color value in their indices of array *color*. All the forbidden columns in $\mathcal{F}$ are moved to $\mathcal{U}$. So $\mathcal{F} = \phi$ and degree of columns of $\mathcal{X}$ in $\mathcal{F}$ is zero. So *u_queue* is reinitialized. As the the selected grouped columns get removed from the *priority_queue* their neighbors' degree lists also get changed.

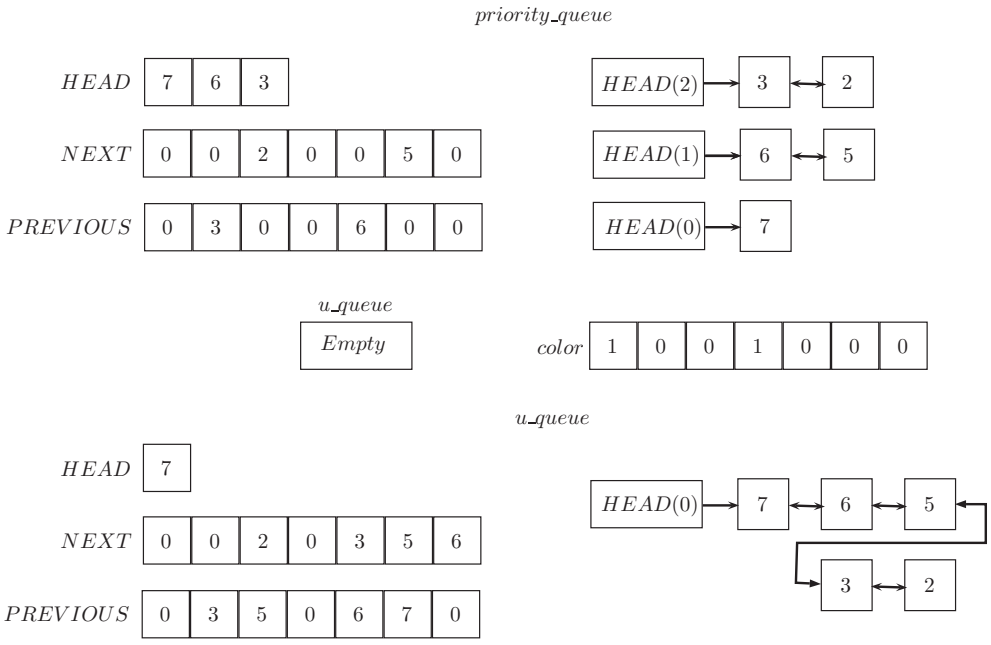

Figure 3.33: Buckets after second iteration in RLFPartition

In the next iteration a new group 2 is created. A column is chosen with maximum degree from *priority_queue*. It is column 3. This column is added to the new group 2 and using the similar process the algorithm updates the degree list of *u_queue*.

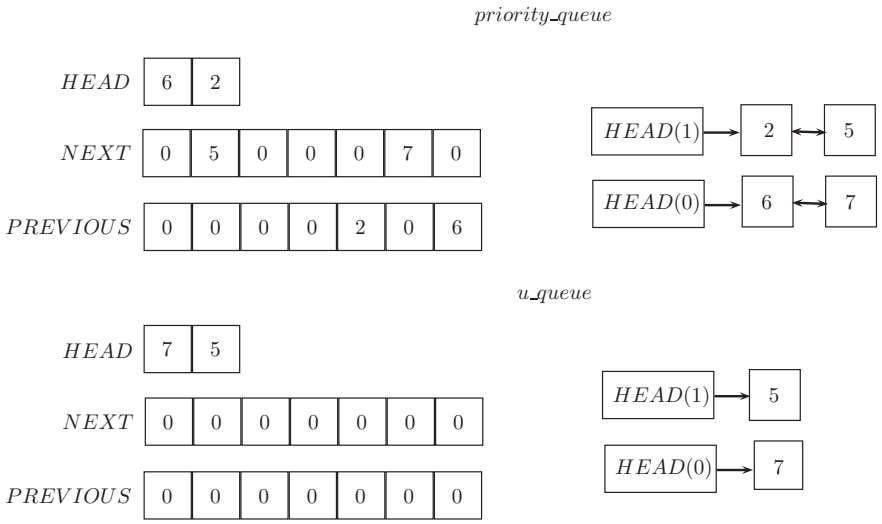

Figure 3.34: Buckets after third iteration in RLFPartition

As *u_queue* is not empty column 5 is chosen from its maximum degree list.

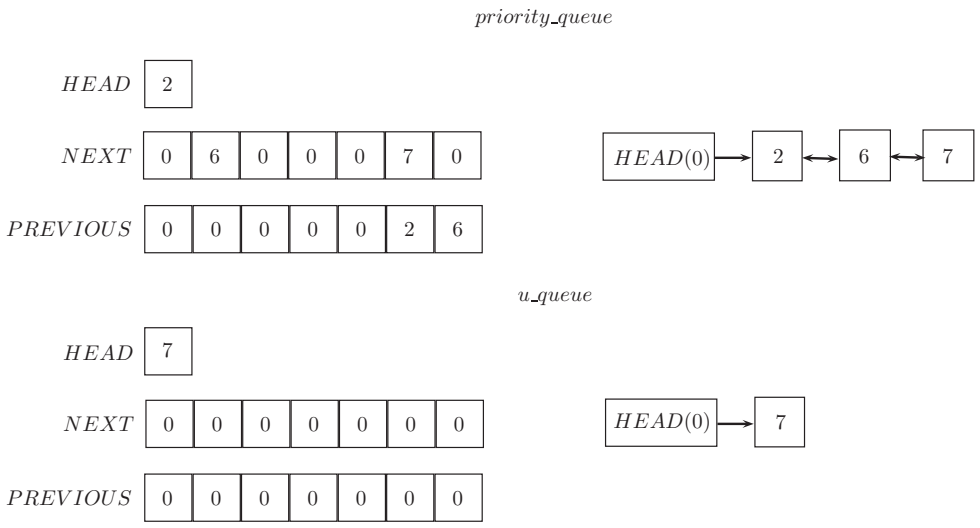

Figure 3.35: Buckets after fourth iteration in RLFPartition

u_queue is not still empty. Column 7 is selected from it. After that *u_queue* gets empty. So we have three columns in structurally orthogonal group 2. So color 2 is assigned as column 3, 5 and 7's color in array *color*. All ungrouped columns from forbidden set then moved to $\mathcal{U}$ and *u_queue* is also reinitialized.

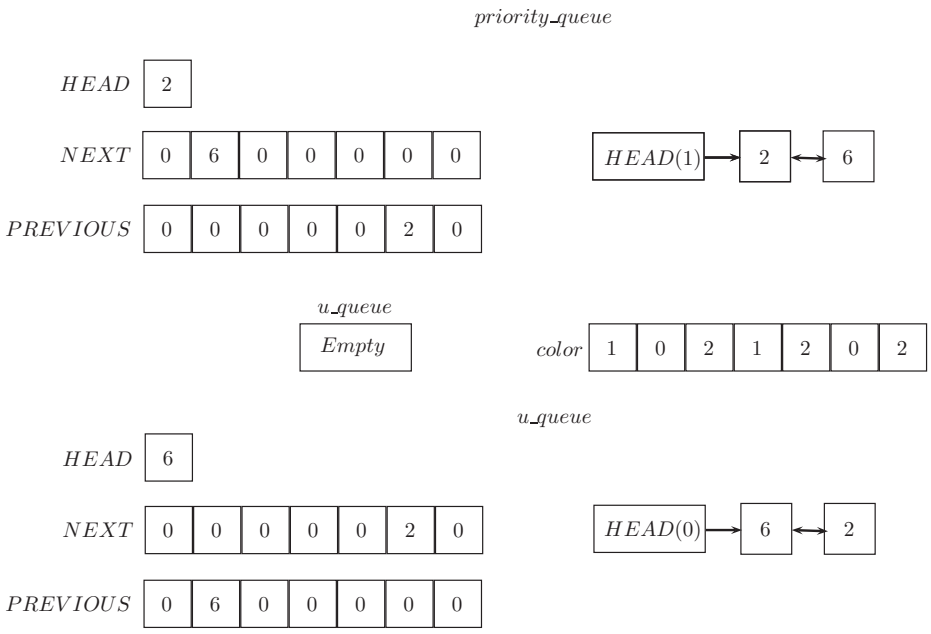

Figure 3.36: Buckets after fifth iteration in RLFPartition

In the next step a new group 3 is formed and column 2 is selected from *prority_queue*

and in the following iteration column 6 is chosen from *u_queue* which is the last column to process. Both column 2 and 6 gets color 3.

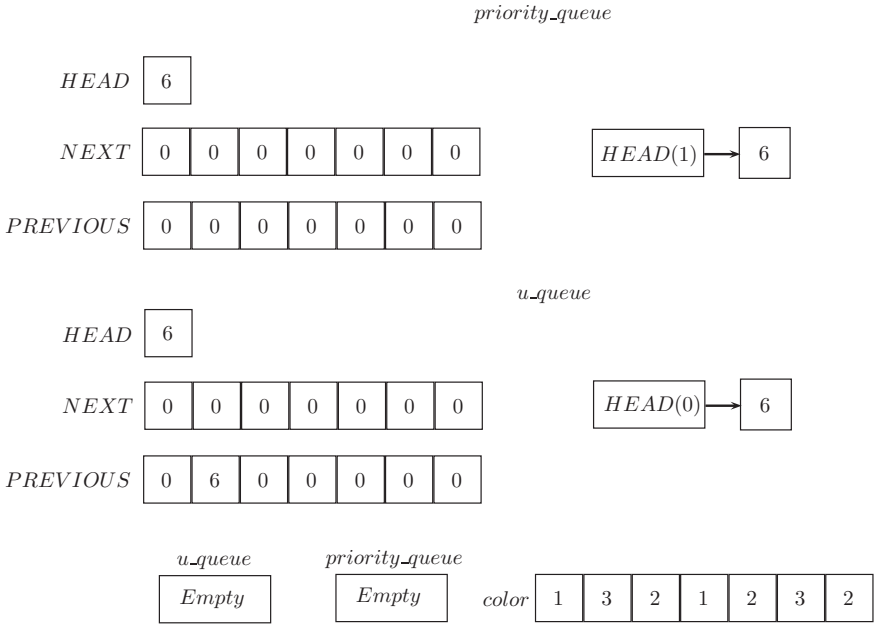

Figure 3.37: Buckets after sixth and final iterations in RLFPartition

The operations in lines 10-15 of algorithm in Figure 3.30 is quite similar to SLO. A column with maximum degree is selected and the algorithms looks for its neighbors in line 11 and do other constant time operation in lines 12-15. We can say that this portion (lines 10-15) requires $O(\sum_{i=1}^{m} \rho_i^2)$ operations and the algorithm repeats these operations *gnum* times. *gnum* is the no. of colors. So overall the running time of RLFPartition is proportional to no. of colors $\times$ $O(\sum_{i=1}^{m} \rho_i^2)$.

3.3 Comparison Between Bucket Heap and Fibonacci Heap

Numerical experiments on selected test instances [16] has been done to compare ordering and partitioning clock time of bucket heap and Fibonacci heap implementations. In Table 3.1 the clock time of the ordering algorithms (SLO and IDO) is compared for bucket heap and Fibonacci heap implementations. The experiments were performed using a PC with 3.4 GHz Intel Xeon CPU, 8 GB RAM, 32 KB L1, 256 KB L2 and 8 MB L3 cache

running Linux.

In Table 3.1 *m*,*n* and *nnz* denote the number of rows, columns and non-zero entries respectively in test matrix. SLO(BH) and SLO(FH) denote clock time of Smallest-Last order using bucket heap and Fibonacci heap. IDO(BH) and IDO(FH) denote clock time of Incidence-Degree order using bucket heap and Fibonacci heap. In most of the cases, bucket heap implementation is faster than the Fibonacci heap implementation. In SLO between these two implementations BH is up to 4.3 times faster but in IDO the difference in running time is more pronounced (BH is up to 10 times faster). For each test file each test was run five times and the each clock time shown in the table is the average time in seconds of 5 test runs.

Table 3.1: Clock time comparison of the Bucket heap and Fibonacci heap in ordering algorithms

Matrix Name	***m***	***n***	***nnz***	**SLO (BH)**	**SLO(FH)**	**IDO(BH)**	**IDO(FH)**
af23560	23560	23560	484256	0.022	0.028	0.024	0.112
cage11	39082	39082	559722	0.044	0.054	0.046	0.278
cage12	130228	130228	2032536	0.186	0.234	0.186	1.154
e30r2000	9661	9661	306356	0.02	0.016	0.02	0.078
e40r0100	17281	17281	553956	0.036	0.036	0.038	0.136
lhr10	10672	10672	232633	0.018	0.014	0.018	0.054
lhr14	14270	14270	307858	0.02	0.022	0.026	0.074
lhr34	35152	35152	764014	0.05	0.056	0.066	0.176
lhr71c	70304	70304	1528092	0.098	0.12	0.128	0.352
lpcrea	3516	7248	18168	0.002	0.006	0.004	0.028
lpcreb	9648	77137	260785	0.188	0.186	0.212	2.142
lpcred	8926	73948	246614	0.188	0.186	0.216	2.122
lpdfl001	6071	12230	35632	0.006	0.008	0.004	0.04
lpken11	14694	21349	49058	0.004	0.012	0.006	0.048
lpken13	28632	42659	97246	0.01	0.024	0.014	0.124
lpken18	105127	154699	358171	0.072	0.16	0.108	0.874
lpmarosr7	3136	9408	144848	0.01	0.012	0.018	0.07
lppds10	16558	49932	107605	0.006	0.026	0.012	0.082
lppds20	33874	108175	232647	0.02	0.07	0.03	0.194
lpstocfor3	16675	23541	76473	0.004	0.01	0.002	0.02

In Table 3.2 the clock time of the partitioning algorithms (SDPartition and RLFPartition) is compared between bucket heap and Fibonacci heap-based implementations. Like

IDO, in SDPartition the difference in running time between two heap implementations is more pronounced. Bucket heap is faster in RLFPartition as well.

Table 3.2: Clock time comparison of Bucket heap and Fibonacci heap in partitioning algorithms

Matrix Name	***m***	***n***	***nnz***	**SDPartition (BH)**	**SDPartition (FH)**	**RLFPartition (BH)**	**RLFPartition (FH)**
af23560	23560	23560	484256	0.034	0.206	0.64	0.852
cage11	39082	39082	559722	0.064	0.37	0.964	1.436
cage12	130228	130228	2032536	0.272	1.584	4.618	7.788
e30r2000	9661	9661	306356	0.032	0.128	0.712	0.782
e40r0100	17281	17281	553956	0.054	0.232	1.33	1.45
lhr10	10672	10672	232633	0.028	0.1	0.124	0.222
lhr14	14270	14270	307858	0.036	0.126	0.172	0.306
lhr34	35152	35152	764014	0.092	0.314	0.43	0.832
lhr71c	70304	70304	1528092	0.18	0.628	0.882	1.932
lpcrea	3516	7248	18168	0.008	0.044	0.094	0.148
lpcreb	9648	77137	260785	0.358	3.932	14.156	25.292
lpcred	8926	73948	246614	0.366	4.038	13.984	24.988
lpdfl001	6071	12230	35632	0.008	0.056	0.06	0.136
lpken11	14694	21349	49058	0.01	0.08	0.04	0.176
lpken13	28632	42659	97246	0.022	0.226	0.114	0.508
lpken18	105127	154699	358171	0.174	1.51	1.106	6.23
lpmarosr7	3136	9408	144848	0.02	0.116	0.686	0.726
lppds10	16558	49932	107605	0.012	0.12	0.06	0.288
lppds20	33874	108175	232647	0.04	0.288	0.154	0.828
lpstocfor3	16675	23541	76473	0.002	0.034	0.012	0.05

3.3.1 The Basics of Cache Memory

Our next task is to do cache analysis on the bucket and Fibonacci heap-based implementations. Let us discuss the basics of cache memory first. In computer memory hierarchy the faster, smaller, and more expensive cache memory is placed at the top and slower and cheaper main memory is placed at the bottom [33]. Cache is situated between the processor and main memory. Cache store data from main memory that are recently referenced. When a reference is satisfied by the cache it is called a cache hit, if not satisfied it is called a cache miss. Besides cache hits and misses, cache evictions occur when cache removes old and relatively unused data from it. A cache miss can make the processor wait for hundreds to thousands of cycles [22]. Due to the smaller size of faster cache memories, we cannot

store all our data in cache memory so cash misses are obvious but locality of reference (we discussed this in Chapter 1) can be advantageous.

A good program is designed in such a way that it exhibits good locality (temporal and spatial). If we design our program with good locality it will certainly run faster than the programs with poor locality.

3.3.2 Cache Analysis of Bucket and Fibonacci Heap

To find why bucket data structure is faster than other implementations we analyzed the cache operations of the algorithms implemented using Bucket data structure and compared with the Fibonacci heap-based data structure. Cache-friendly data structure of bucket heap makes it faster than the other heap implementation.

The cache complexity $Q(n;Z,L)$ of an algorithm is the number of cash misses it occurs [13]. Here Z is the size of the function, and L is the line length of the ideal cache. By clearing Z and L from the context cache complexity is denoted simply as $Q(n)$. We take two matrices from The University of Florida Sparse Matrix collection[10].

To analyze the cache performance of our implementations we use a cache simulator that was developed as a part of a course project. We developed the simulator with the help of Valgrind framework [28]. The simulator takes Valgrind memory traces as input, then from the traces, it calculates the number of cache hits, misses, and evictions. The memory traces we get from Valgrind look like as follows:

I 0040d7d4, 8

M 0421c7f0, 4

L 04f6b868, 8

S 7ff0005c8, 8

The memory traces from Valgrind starts with “I”, “M”, “L”, and “S” followed by 64-bit hexadecimal memory address field and size field. “I”, “M”, “L”, and “S” denote instruction load, data modify, data load, and data store respectively. Each load/store causes at most one

cache miss. Data modify is a load followed by a store to the same address, it causes two hits, or a miss and hit with a possible eviction. In the input of the simulator, we provide memory trace file we get from Valgrind. We also provide number of set index bits(s), number of lines per set(E) and number of block bits(b) of our cache as inputs. From the number of set index bits and block bits we calculate a number of sets and block size of the cache. The simulator provides the number of cache hits, misses, and eviction as output. The simulator works for any arbitrary cache size. By passing s, E and b we set the size of the cache and the hits, misses, and evictions are calculated based on this cache size.

We calculate cache hits, misses, and evictions for ordering and partitioning algorithms using bucket heap and Fibonacci heap data structures with the help of our simulator. The test environment is same as mentioned in Section 3.3. The PC we use to perform our experiments has an L1 cache of 32 KB. Number of set index bits(s), number of lines per set(E) and number of block bits(b) of our cache are 6, 8, and 6 from which we set size ($2^6 \times 8 \times 2^6$) of the cache to 32 KB. If we look at the cache misses in Table 3.3 and Table 3.4 then we see that in bucket data structure the number of cache misses is much less than the other.

Table 3.3: Matrix name: west0067, $m = 67$, $n = 67$, $nnz = 294$

	Hits		**Misses**		**Evictions**	
	BH	FH	BH	FH	BH	FH
SLO	186730	191584	653	729	156	221
IDO	186731	236167	645	852	148	340
SDPartition	187521	250646	661	911	162	399
RLFPartition	210116	273297	658	818	162	306

Table 3.4: Matrix name: eris1176, m = 1176, n = 1176, nnz = 9864

	Hits		**Misses**		**Evictions**	
	BH	FH	BH	FH	BH	FH
SLO	14605021	14718890	51887	68973	51375	68461
IDO	14625756	17297517	54564	130683	54052	130171
SDPartition	14890594	18412789	71567	157269	71055	156757
RLFPartition	63143315	71284068	319190	703209	318678	702697

3.4 Performance Profile for DSJM and ColPack

We do some numerical experiments on selected test instances [16]. In this section, we provide and discuss the results from the numerical experiments. We compare the performance of DSJM and ColPack [17] in terms of

- Number of structurally orthogonal groups in the partition
- Clock time for selected algorithms

The test environment is same as described in Chapter Section 3.3. In Table 3.5 we display the partitioning results of DSJM and ColPack software. We have already stated the denotation of m, n and nnz in the previous section (3.3). In both of these two software number of ordering and partitioning algorithms are being implemented and here we show the best partitioning results produced by the two software. In 8 test cases, we get different partitioning results. Out of 8 DSJM yields better partitioning on 7 of them.

For the sequential (SEQ) partitioning and ordering implementations of DSJM and ColPack, we compare the running time using performance ratio defined by Moré et al. [11]. If total number of problems is n_p and total number of solvers is n_s (in our case it is 2, DSJM and ColPack) then for each problem p and solver s, $t_{p,s}$ can be defined as follows

$$t_{p,s} = \text{clock time required to solve problem } p \text{ by solver } s$$

Then performance ratio can be defined as

Table 3.5: Partitioning results

Matrix Name	***m***	***n***	***nnz***	**ColPack**	**DSJM**
af23560	23560	23560	484256	42	**38**
cage11	39082	39082	559722	64	**54**
cage12	130228	130228	2032536	67	**56**
e30r2000	9661	9661	306356	68	**65**
e40r0100	17281	17281	553956	66	66
lhr10	10672	10672	232633	63	63
lhr14	14270	14270	307858	63	63
lhr34	35152	35152	764014	63	63
lhr71c	70304	70304	1528092	63	63
lpcrea	3516	7248	18168	360	360
lpcreb	9648	77137	260785	845	**844**
lpcred	8926	73948	246614	808	808
lpdfl001	6071	12230	35632	228	228
lpfit2d	25	10524	129042	10500	10500
lpken11	14694	21349	49058	124	**122**
lpken13	28632	42659	97246	171	**170**
lpken18	105127	154699	358171	325	325
lpmarosr7	3136	9408	144848	**70**	76
lppds10	16558	49932	107605	96	96
lppds20	33874	108175	232647	96	96
lpstocfor3	16675	23541	76473	15	15

$$r_{p,s} = \frac{t_{p,s}}{\min\{t_{p,s} \mid s \in \{DSJM,\ ColPack\}\}}$$

We calculate $r_{p,DSJM}$ and $r_{p,ColPack}$ for each problem from Table 3.5 . We do not only compare the performance ratio of any specific problem for two solvers instead we would like to assess the overall performance of the solvers. For that purpose we use

$$\rho_s(\tau) = \frac{1}{n_p}(number\ of\ problems\ on\ which\ r_{p,s} \leq \tau)$$

By varying the speed up factor τ and by calculating their corresponding $\rho_{DSJM}(\tau)$ and $\rho_{ColPack}(\tau)$ we get the performance profile of the solvers.

The performance profile on benchmark instances for IDO + SEQ coloring time is shown in Figure 3.38

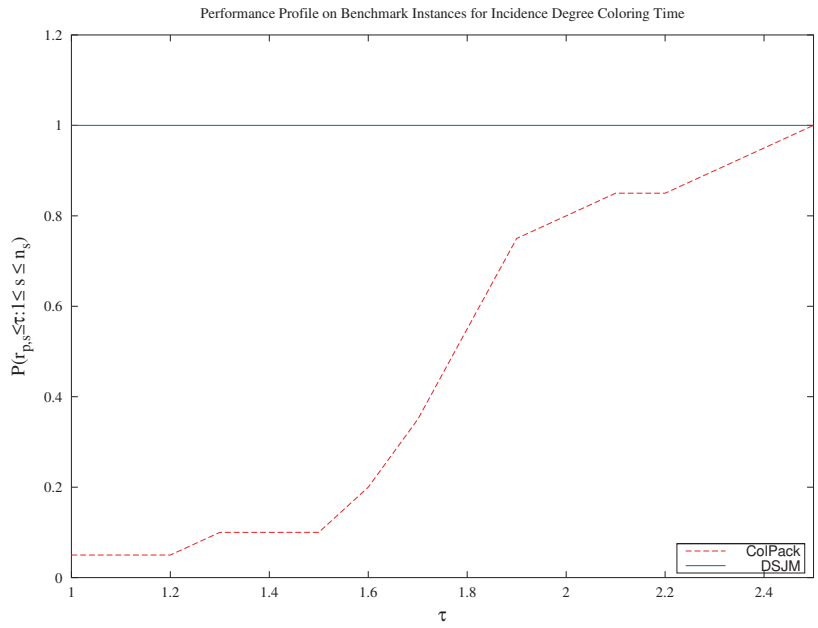

Figure 3.38: Performance profile for sequential partitioning with IDO

The performance profile on benchmark instances for SLO + SEQ coloring time is shown in Figure 3.39

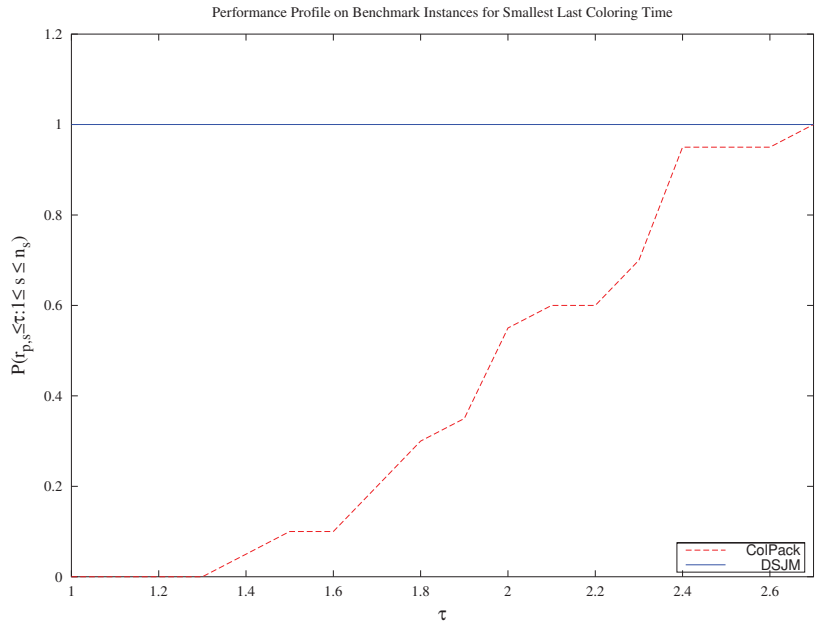

Figure 3.39: Performance profile for sequential partitioning with SLO

The performance profile on benchmark instances for Largest-First Ordering(LFO) +

SEQ coloring time is shown in Figure 3.40

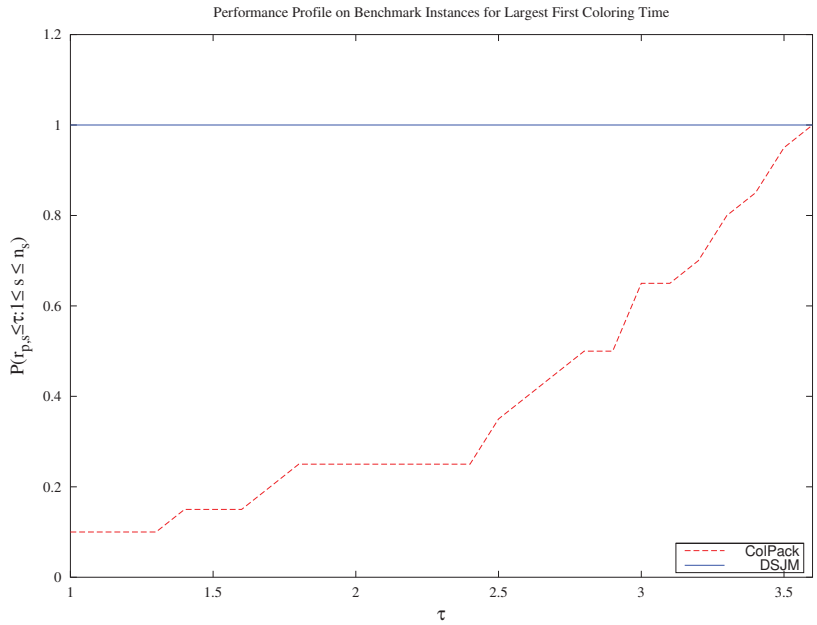

Figure 3.40: Performance profile for sequential partitioning with LFO

The clock time here is the combined time of ordering and sequential partitioning algorithm for the problems in Table 3.5. From the figures, we see that DSJM is faster in all cases compared to ColPack. If we use very large-scale problems to compare running time of the two software, it is expected to be more pronounced.

Chapter 4

Efficient Implementation of Exact Graph Coloring Algorithms

In this chapter, we study exact graph coloring methods. There are few exact graph coloring approaches available. Classical integer programming models can be one approach. For example, Mehrotra and Trick proposed a Branch and Price algorithm [27] based on set covering formulation. If S is the set of all independent sets and each independent set $s \in S$ is associate with a binary variable x_s, which is 1 if and only if the vertices of set s gets the same color. The model is given below.

$$\min \sum_{s \in S} x_s, \tag{4.1}$$

$$\sum_{s \in S : i \in S} x_s \geq 1 \quad i \in V, \tag{4.2}$$

$$x_s \in \{0,1\} \quad s \in S. \tag{4.3}$$

The objective function 4.1 minimizes the number of independent sets so the total number of colors is minimized. The constraints of 4.2 state, each vertex must be in at least one independent set. Here V is the set of all vertices. Constraints of 4.3 state, variable x_s must be binary.

Brélaz [3] proposed DSATUR algorithm based exact graph coloring algorithm by modifying Randal-Brown's algorithm [4]. Brélaz's algorithm had errors in two steps which was corrected by Peemöller [30]. We consider Brélaz's DSATUR based exact coloring approach as it has useful structures that can be exploited when formulated and implemented

as a matrix coloring problem. This exact algorithm recursively divides a graph coloring instance into series of subproblems. In each subproblem, the graph is partially colored. While coloring the graph there is an upper bound (*UB*). Initially the *UB* can be the number of columns because we may need at most this number of colors. If the graph is partially colored and the number of colors used is already greater than UB then there is no need to go forward. In another word, we can say that we prune the search tree. This pruning mechanism makes this algorithm a branch-and-bound algorithm. If at any point every vertex of the graph is colored with k colors and k is less than *UB* then the *UB* is updated and set to k. New subproblems are created recursively if the graph is partially colored and $k < UB$. When a vertex is selected for coloring, each available color in k is assigned to that vertex and new subproblem is formed for each available color assigned to that vertex. Another subproblem is formed by assigning $k+1$ color to the selected vertex if the color is available and $k+1 <$ *UB*. A clique of the graph is first partially colored. The size of this clique is implicitly used as lower bound (*LB*).

A clique is subgraph which is complete, i.e., every pair of vertices are connected by an edge. We set *LB* to size of a clique because we need at least this number of colors. The algorithm terminates when either there are no subproblems left or *UB=LB*.

Choosing a vertex for coloring is crucial and has a large effect on the algorithm. Based on Brélaz's DSATUR algorithm a vertex is chosen which has a maximum number of differently colored neighbors. The reason behind choosing such kind of vertex is it reduces the number of subproblems created at each branch because the vertex with maximum differently colored neighbors has less available colors than other vertices.

A widely used enumeration exact coloring code first finds a maximal clique, assigns colors to its vertices, and then performs a branch-and-bound type search with back-track [27]. The graph is represented by its adjacency matrix. In this chapter we give a branch-and-bound type exact algorithm for finding a structurally orthogonal partition of the columns of a sparse matrix $A \in \mathbb{R}^{m \times n}$. Our algorithm has a number of distinct features compared

with the publicly available code [27]. First, in the matrix coloring problem, each row i for $i = 1,2,\ldots,m$ the columns $\{j|a_{ij} \neq 0\}$ are mutually structurally dependent and therefore defines a clique in graph-theoretic sense. Thus a maximal clique is obtained by identifying a row with a maximum number of nonzero entries which is denoted by ρ_{max}. Thus, our lower bound calculation is straight forward. Second, in our computer implementation, we employ compressed sparse data structures to represent the sparsity pattern of the matrix. This ensures that larger problem instances can be represented and handled. Additionally, we utilize a bucket data structure to choose the next vertex to be colored [20]. We explore several chromatic degree-based vertex selection approaches and alternative tie-breaking strategies. Our exact coloring implementation can be used independently or as a subroutine to color a small critical submatrix in a combined approach where the partial coloring is extended to the entire matrix using heuristics[24].

4.1 The Algorithm

We have already generally described how DSATUR based branch-and-bound type exact graph coloring algorithm works above. Now we describe the algorithm in detail. At first, we find a clique to find the lower bound *LB*. In our case it is ρ_{max}. Then the member columns of ρ_{max} are colored and they will not be recolored again. While coloring the member columns of ρ_{max} the saturation degrees of the adjacent columns are updated. After coloring the clique the *eaxctColor* method is called to color rest of the uncolored columns. Figure 4.1 shows the our exact graph coloring algorithm. Let us describe the algorithm in detail.

1. **Termination and backtracking conditions:** The problem/subproblems of the algorithm terminate based on the conditions in lines 1-4. There are three conditions. If the current $colorBoundary \geq UB$ then the current call terminates. It means the *colorBoundary* is already greater or equal to the upper bound so we can terminate and return from here without searching for new subproblems. Another termination

condition is when $order = N$. It means there are no subproblems left. The last termination condition is *UB*=*LB*. When *UB*=*LB* we have already reached to the minimum color, so we terminate and return.

2. **Column selection:** Column selection is one of the crucial things of this algorithm. In lines 5-6 we select a column. In section 4.3, we will discuss in detail how we select a column for coloring. But now for simplicity's sake, let us assume a column with maximum saturation degree gets selected. If there are more than one columns of maximum saturation degree, then the column that appears first in the bucket is selected. The *handled* tag is set to true for the selected column. This *handled* tag is used when we update the saturation degrees of columns.

3. **Coloring of selected column and subproblems formulation:** Coloring the selected column and other critical operations are done in lines 7-23. Each color from 1 to *colorBoundary* is tested on the selected column *jcol* to see whether they are available for *jcol* or not. The minimum available color *i* is assigned to *jcol* in line 9. Then the column gets deleted from the bucket because only uncolored vertices are kept in the bucket. In lines 11-13 the maximum saturation is updated, and new color class is created if the available color *i* is greater than the current maximum saturation degree. Then a method name *satDegInc* is called. Using this method, we efficiently increase the saturation degrees of the adjacent vertices of the colored column. We will describe this method later. A new subproblem is then formed that recursively colors the uncolored columns and returns the total colors required. We call this $updatedColring$. If this $updatedColring < UB$ then the *UB* is updated in line 17. The assigned color is then removed from *jcol*, and is added back to the bucket. As *jcol* is moved from colored set of vertices to uncolored set of vertices, the saturation degree of its neighbors are needed to be updated. The *satDegDec* method in line 18 does this task. The upper bound *UB* was updated in line 22 and if this $UB < colorBoundary$ then the current call terminates and returns the new updated *UB*. The operations in between lines

8-23 are done for every available color for $jcol$. So, briefly we can say that, for each available color, $jcol$ is colored and then the rest of the uncolored columns are colored recursively, and this is how the algorithm looks for new and improved coloring.

4. **Another new subproblem formulation:** From lines 24-36 another new subproblem is formulated. $colorBoundary$ indicates the number of colors in the current best partial coloring. If $colorBoundary + 1 < UB$ then $colorBoundary + 1$ numbered color is assigned to $jcol$. Then the rest of the uncolored columns are colored recursively. The remaining operations done here are same as described above.

4.1.1 Updating Saturation Degree:

Saturation degree updating is an important task in our exact graph coloring algorithm. We select a column for coloring from maximum saturation degree set which has a large effect on the number of subproblems explored. With the help of CSR, CSC and bucket data structure we can update the saturation degree quite efficiently. CSR and CSC are used to look for the adjacent columns of a selected column and bucket data structure is used to update the degree lists (in this case it is saturation degree) of the uncolored vertices. In our exact coloring algorithm we both need to increase and decrease saturation degrees of the columns. When a column is colored the saturation degrees of its adjacent columns are required to be increased, and when a color is removed from a column, then saturation degrees of its adjacent columns are needed to be decreased if applicable. In Figure 4.2 the mechanism of increasing saturation degree is shown and in Figure 4.3 the mechanism of decreasing saturation degree is shown.

1. **Increasing saturation degree:** Figure 4.2 shows the method for increasing saturation degrees. When a column is selected and colored the adjacent columns are inspected. If the colored column is a different colored neighbor for any adjacent column, then the saturation degree of that column is increased. A two-dimensional array named

```
EXACTCOLOR(order, colorBoundary)
 1  if colorBoundary ≥ UB or order = N
 2      return colorBoundary
 3  if UB = LB
 4      return UB
 5  jcol ← getColumn()
 6  handled[jcol] ← true
 7  for each color i from 1 to colorBoundary
 8      if colorAvailable(jcol, i)
 9          color[jcol] ← i
10          deleteColumn(head, next, previous, satDeg[jcol], jcol)
11          if i > maximumSaturation
12              maximumSaturation ← maximumSaturation + 1
13              createNewColorClass()
14          satDegInc(jcol, color[jcol])
15          updatedColoring ← exactColor(order + 1, colorBoundary)
16          if updatedColoring < UB
17              UB ← updatedColoring
18          satDegDec(jcol, color[jcol])
19          color[jcol] ← N
20          addColumn(head, next, previous, satDeg[jcol], jcol)
21          if UB <= colorBoundary
22              handled[jcol] ← false
23              return UB
24  if colorBoundary + 1 < UB
25      color[jcol] ← colorBoundary + 1
26      deleteColumn(head, next, previous, satDeg[jcol], jcol)
27      if colorBoundary + 1 > maximumSaturation
28          maximumSaturation ← maximumSaturation + 1
29          createNewColorClass()
30      satDegInc(jcol, color[jcol])
31      updatedColoring ← exactColor(order + 1, colorBoundary + 1)
32      if updatedColoring < UB
33          UB ← updatedColoring
34      satDegDec(jcol, color[jcol])
35      color[jcol] ← N
36      addColumn(head, next, previous, satDeg[jcol], jcol)
37  handled[jcol] ← false
38  return UB
```

Figure 4.1: DSATUR based exact graph coloring algorithm

colorTracker is used to check whether the colored column is a different colored neighbor. It is quite same as the bitset utilized in the SDPartition algorithm. For each color a row of size N is created in *colorTracker*. We call it creating a new color class. It is done in Figure 4.1's lines 13 and 29. Suppose $colorTracker[4][5] = 0$. It means column 5 does not have any neighbor that is colored using a color no. 4. So if a neighbor of column 5 is colored using a color no. 4 then column 5 has a uniquely colored neighbor, so the saturation degree of column 4 is increased as well as the value of $colorTracker[4][5]$. This *colorTracker* array helps us to update the saturation degree of any column in constant time because for this we do not need to look for the adjacent of adjacent columns of the colored column which is an expensive operation. The inputs of this method are *jcol* and its *colorNo*. Efficient tagging scheme is used to update saturation degrees. For example, we don't need to update saturation degree of *jcol* because it is already colored rather we look for its neighbors. So $tag[jcol]$ is set to true in line 2. While looking for adjacent columns of *jcol* what we do is take one non-zero element of *jcol* and traverse through other non-zero elements in the same row. Once any nonzero element of any column is traversed, that column is tagged and will not be considered in any next iterations. In line 6 we get $colorTracker[colorNo][j]$ and if it is 0 then we update the saturation degree of column j. While updating saturation degree, all colored and uncolored columns are considered but we update degree lists of the columns in the bucket which are uncolored in lines 9-11. The reason we update saturation degree of all columns is, in exact coloring algorithm we color a column and remove the color to construct new subproblem. When color is removed from a column it i s added back to the bucket based on its saturation degree. For this reason, we update saturation degrees of both colored and uncolored columns. In line 12 the value of $colorTracker[colorNo][j]$ is increased either the saturation degree of j is increased or not. In SDParitition we only need the information about if a column j has *colorNo* colored neighbor or not to

increase saturation degree but here we need the count of *colorNo* colored neighbors of j because it is important when we decrease the saturation degree of a column.

```
SATDEGINC(jcol, colorNo)
 1  intializeTag()
 2  tag[jcol] ← true
 3  for all j ∈ adj(jcol)
 4      if tag[j] = false
 5          tag[j] ← true
 6          colorCount ← colorTracker[colorNo][j]
 7          if colorCount = 0
 8              satDeg[j] ← satDeg[j] + 1
 9              if handled[j] = false
10                  deleteColumn(head, next, previous, satDeg[j] − 1, j)
11                  addColumn(head, next, previous, satDeg[j], j)
12          colorTracker[colorNo][j] ← colorTracker[colorNo][j] + 1
```

Figure 4.2: Update(Increase) saturation degree

2. **Decreasing saturation degree:** Figure 4.3 shows the method for decreasing saturation degrees. When a color is removed from a column *jcol* its added back to uncolored set of vertices which means the saturation degrees of its adjacent columns are needed to be updated. If for any $j \in adj(jcol)$, *jcol* is the only *colorNo* colored neighbor then we will get $colorTracker[colorNo][j] = 1$. Removing *colorNo* from *jcol* means there is no *colorNo* colored neighbor for j so the saturation degree of j is decreased in line 8 and degree list of j is update in bucket in lines 9-11. $colorTracker[colorNo][j] = 1$ or not, either way the value of $colorTracker[colorNo][j]$ is decreased in line 12 as now j has one less *colorNo* colored neighbor.

4.2 An Example

For a better understanding how the algorithm works a simple example is given below. We choose a matrix of 20 rows and 30 columns with 60 non-zero entries. The Matrix B is shown in Figure 4.4. We can associate a graph with this matrix. Each column of this matrix

```
SATDEGDEC(jcol, colorNo)
 1  intializeTag()
 2  tag[jcol] ← true
 3  for all j ∈ adj(jcol)
 4      if tag[j] = false
 5          tag[j] ← true
 6          colorCount ← colorTracker[colorNo][j]
 7          if colorCount = 1
 8              satDeg[j] ← satDeg[j] − 1
 9              if handled[j] = false
10                  deleteColumn(head, next, previous, satDeg[j] + 1, j)
11                  addColumn(head, next, previous, satDeg[j], j)
12          colorTracker[colorNo][j] ← colorTracker[colorNo][j] − 1
```

Figure 4.3: Update(Decrease) saturation degree

is a vertex of the graph. If there are non-zero elements in the same row, then the vertices are connected by edges. Suppose if we look at row 20 of column 1, 2 and 3 we see that they have nonzero entries in the same row. So these three columns are connected by edges. Figure 4.5 shows the column intersection graph $G(B)$ of matrix B.

The first thing the algorithm does is to find a clique to determine a *LB*. In our implementation we use ρ_{max} as *LB*. ρ_{max} for matrix B is 3. So initially a clique of size three is colored. In matrix B there are more than one clique of size 3. We find ρ_{max} and the index of any one the ρ_{max}-clique at the time we compute the degrees of the columns. For matrix B the vertices of clique initially colored are column 2, 19 and 20. These columns are colored and will never be recolored. The saturation degrees of the adjacent columns are updated in the mean time. After coloring the clique The $exactColor(order, colorBoundary)$ method is called. Now we discuss in detail with examples how this method works.

For matrix B, after coloring the clique the value of *order* and *colorBoundary*, are both three because three columns are colored, and three colors were used to color the clique. So $exactColor(3,3)$ will be called. When the $exactColor(order, colorBoundary)$ is called for the first time, the upper bound *UB* is no. of columns, in this case, it is 30, and lower bound

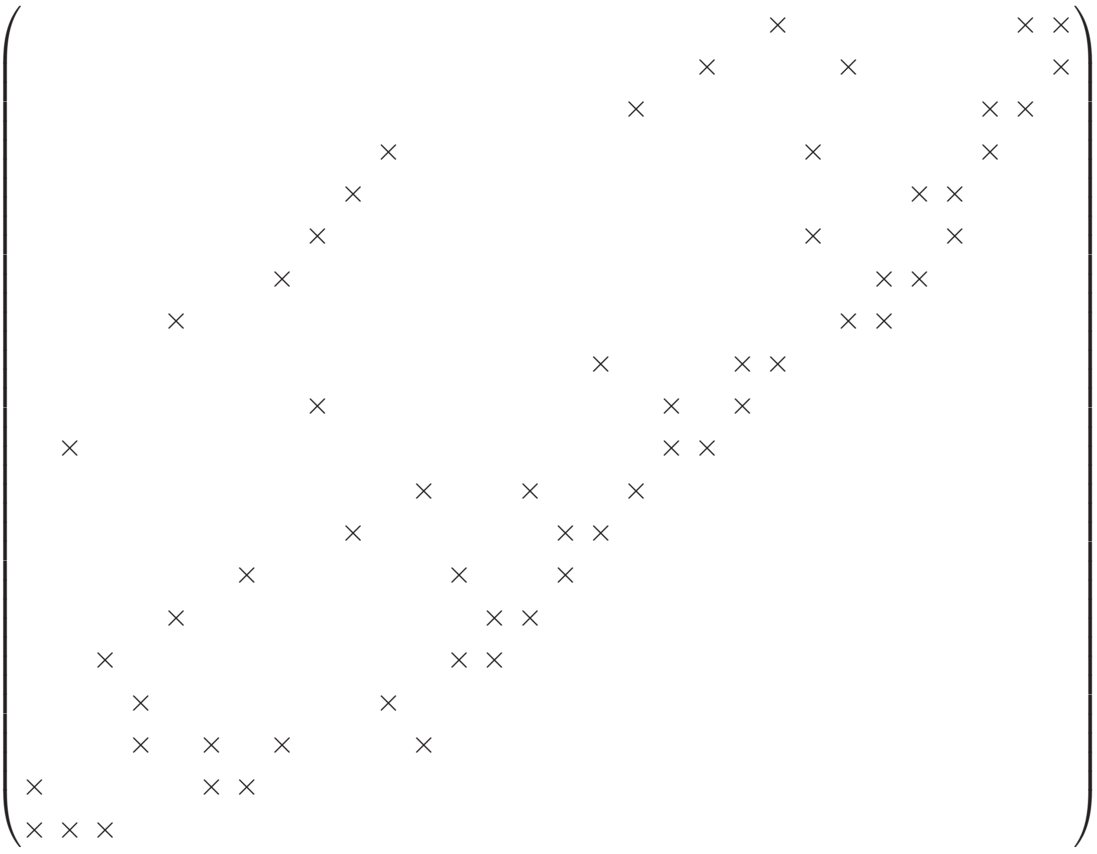

Figure 4.4: Matrix B

LB is 3. The *LB* will always be three, but *UB* will be updated. In Figure 4.6 the steps of the operations are described. After $exactColor(3,3)$ is being called a column with maximum saturation degree gets selected in line 5 of algorithms of Figure 4.1. Column 30 is selected. Now we have up to 29 options to color column 30. So for column 30 only it is possible to create up to 29 new subproblems.

In lines 7-23 of the algorithm of Figure 4.1 new subproblems are formed depending on available colors for the selected column between color 1 to *colorBoundary*. Color 1 is available for column 30 so color one is assigned to column 30 saturation degrees of its neighbors are updated then a new subproblem is recursively created hence $exactColor(4,3)$ is called. Three columns of the cliques and one new column 30 has been colored, so the value of *order* is four now. *order* tracks how many columns have been colored.

For each subproblem, possible branches are 29 now, but it does not expand all branches.

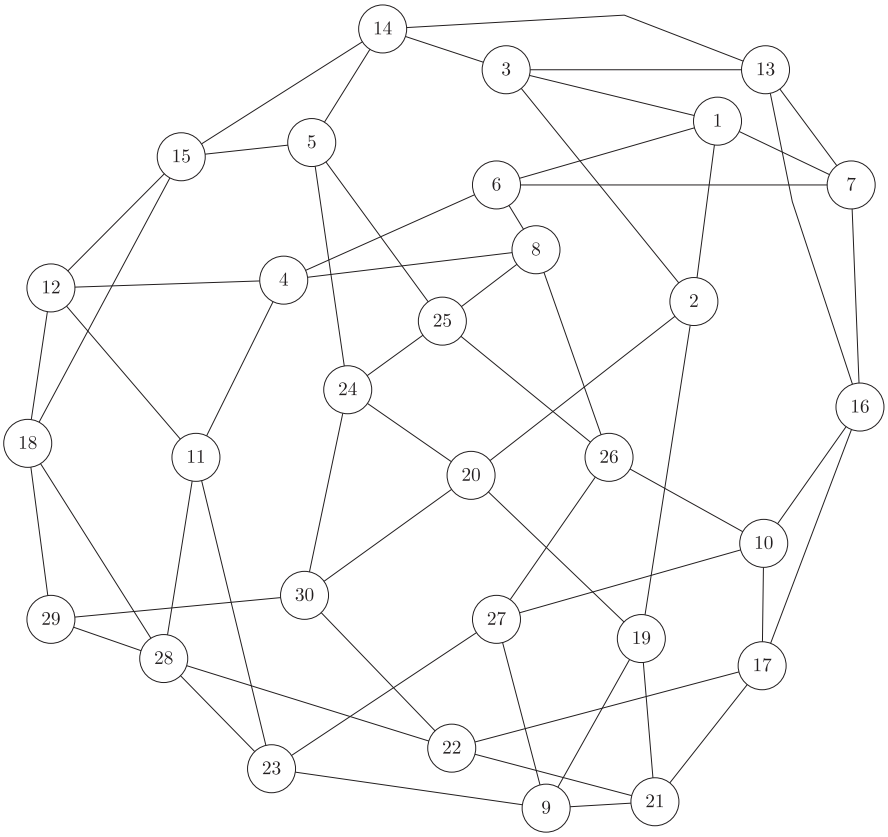

Figure 4.5: Column intersection graph $G(B)$ of matrix B

For example in $exactColor(4,3)$ the selected column is 24. Here it is not possible to expand a branch by assigning color 1 to column 24 because one of its neighbors has already been colored using color 1. So a new subproblem is formulated by recursively calling $exactColor(5,3)$. Figure 4.6 shows how a new subproblem is formed at each step after assigning an available minimum color to the selected column.

If we look at $exactColor(25,3)$ of Figure 4.6 we see that color four has been assigned to the selected column 28. It is because neither color 1, 2 or 3 are available for 28. So the algorithm checks if $colorBoundary + 1 < UB$ (line 24). In this case, it is true. So $colorBoundary + 1 = 4$ numbered color is assigned to column 28. As the color assigned is greater that the current *colorBoundary*, so it is updated and new *colorBoundary* is 4. In $exactColor(29,4)$ column 21 is selected and colored, and it is the last column to color. So when $exactColor(30,4)$ is called in line 1 of Figure 4.1 the $order = N$ condition is true so

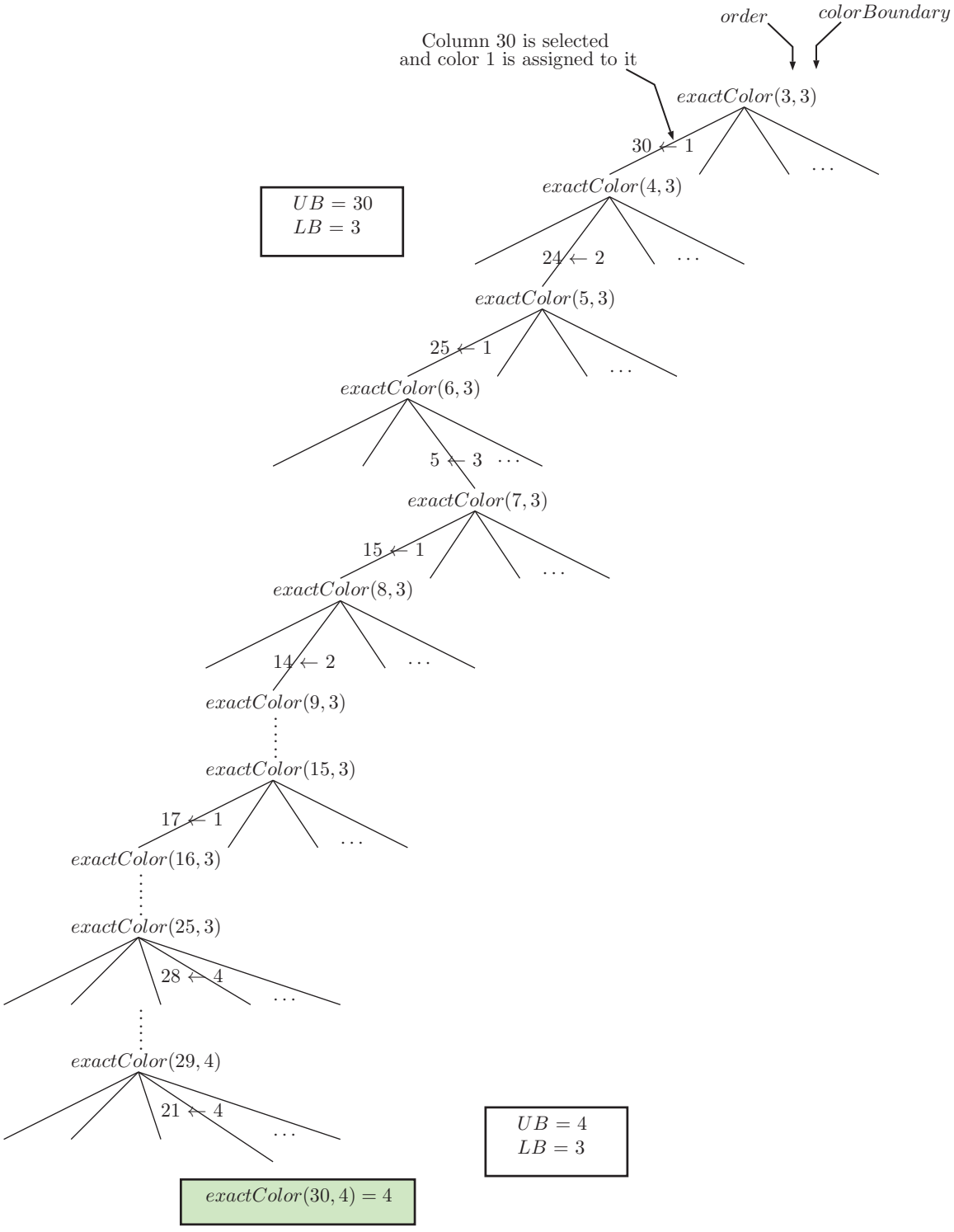

Figure 4.6: *exactColor*(*order*, *colorBoundary*) steps

the algorithm does not proceed instead it returns from here. So from the sequence shown in Figure 4.6, we get that the graph can be colored using four colors. The coloring of the

graph is shown in Figure 4.7. The mapping of numbers to their corresponding colors for Figure 4.7 is given bellow.

$$1 \leftarrow red$$

$$2 \leftarrow blue$$

$$3 \leftarrow green \text{ and}$$

$$4 \leftarrow yellow$$

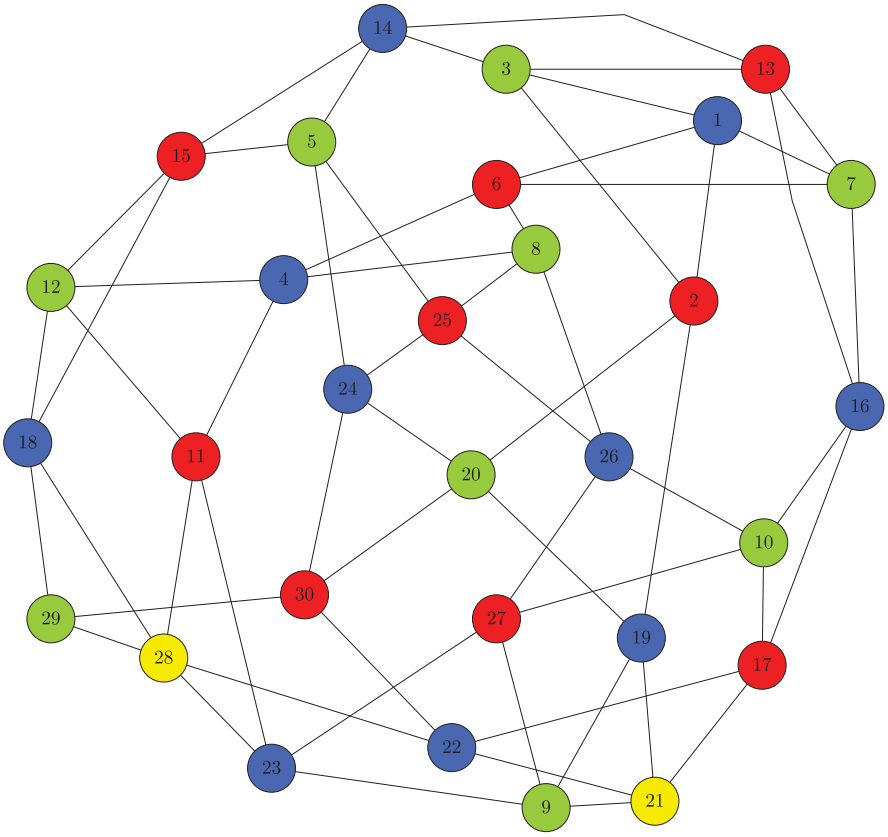

Figure 4.7: Coloring of $G(B)$ using 4 colors

As we have got a new coloring, we update the *UB*. *UB* is set to 4. Till now for every subproblem, it was possible to form up to 29 branches or subproblems. But now as the *UB* has been updated, so it is possible to form up to 3 subproblems only. At this stage, we are not sure whether 4 is the optimal coloring for $G(B)$ or not. The *LB* is 3, and we have not explored many branches yet. So the algorithm will start to backtrack from this point with

new *UB*, which has been set to 4. So the algorithm will backtrack and will try to explore new branches by assigning either color 1, 2 or 3 if available.

Figure 4.8 shows the backtracking. The algorithm backtracks from $exactColor(30,4)$ and goes to $exactColor(29,4)$. While backtracking the assigned color gets removed from the selected column (lines 18-20 and 34-36 of Figure 4.1) as The *UB* has been changed and now we explore new branches by assigning a new available color to any selected column. When a color is removed from any column, it is added back to the bucket, and the saturation degrees of its adjacent columns are also updated. In $exactColor(29,4)$ it looks for if there is any available color between 1-3 for column 28. There is no available color, so it will continue to backtrack. At each step, the algorithm will look if it is possible to expand a new branch. The algorithm will continue to backtrack until it reaches $exactColor(15,3)$ because it was not feasible to open any new branch in between. In $exactColor(15,3)$ column 17 was already colored using color 1 . Now we see that color 3 is also available for column 17. So color 3 is assigned to column 17 a new subproblem is formed. Then in the next step in $exactColor(16,3)$, column 10 is selected, and color 1 is assigned to it. Branching is continued until the algorithm reaches $exactColor(26,3)$. In $exactColor(26,3)$ column 21 is selected but neither color 1, 2 or 3 is available for column 21. So for this branching, unfortunately, we do not get any feasible solution. There is no necessity to go forward, thus the algorithm backtracks again.

Figure 4.9 show backtracking from $exactColor(26,3)$ and new branching. The algorithm backtracks and in each step check whether it is possible to open a new branch. As it is not possible to open any branch, it continues to backtrack until it reaches $exactColor(7,3)$. A branch was already expanded by assigning color 1 to column 15. Now we see that it is possible to assign color 3 to column 15. So a new subproblem is created. The algorithm goes to $exactColor(8,3)$. In $exactColor(8,3)$ column 14 is selected and color 1 is assigned to it. The algorithm continues to create new subproblems and opens new branches. In $exactColor(29,3)$ column 18 gets color 3 which is the last column to process.

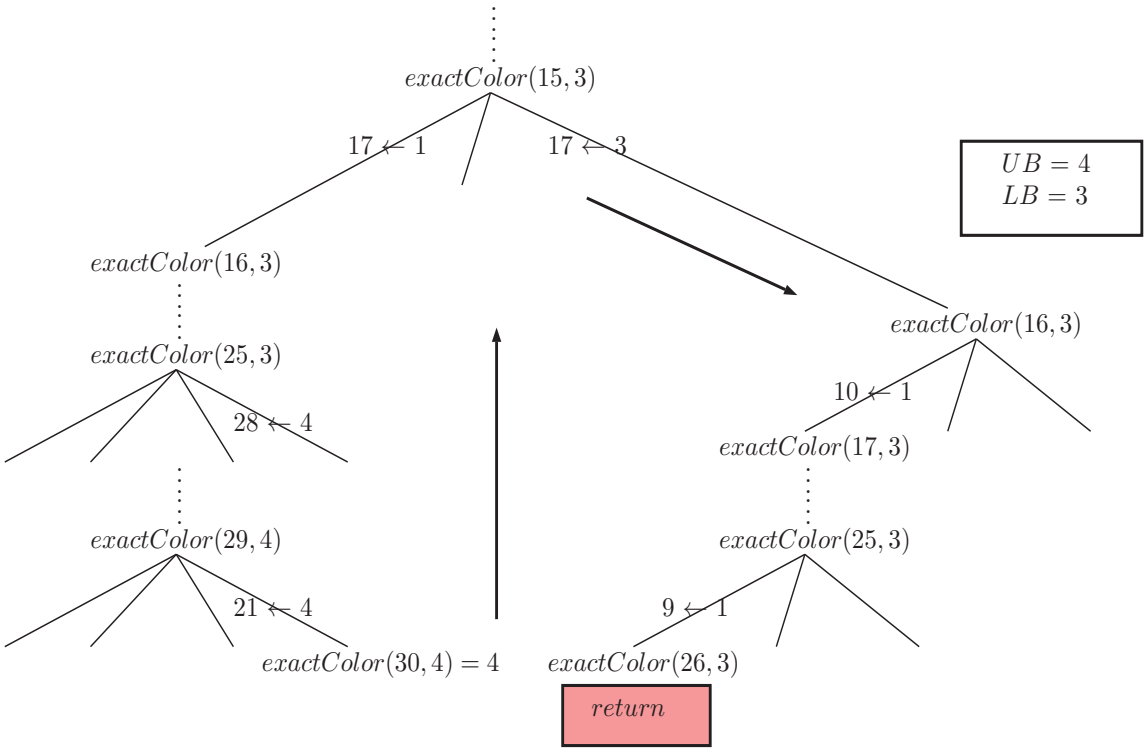

Figure 4.8: Backtracking and branching after getting a feasible coloring

In $brachColor(30, 3)$ we do not have any more columns to color as $order =$ no. of columns , so it returns the total no. of colors 3. This coloring of 3 is our optimal coloring because our *LB* is also 3. We do not need to explore any new branches.

What happens when we do not reach to *LB*? If we do not get any coloring that is equal to *LB*, then we continue to explore new subproblems until none left. Then the lowest feasible coloring we got in between is the optimal coloring. The optimal coloring of $G(B)$ is shown in Figure 4.10.

We have seen that we had many branching options at the beginning. For every search problem, we could form up to 29 search trees. But by selecting the column with maximum saturation degree, we narrow down our choices. A column with higher saturation degree has less available colors, so many search trees are pruned. Then after getting a feasible coloring of 4 colors, we updated the *UB*. This way many more search trees are pruned because at the beginning we had color 1 to 29 as options but after updating the *UB* we had

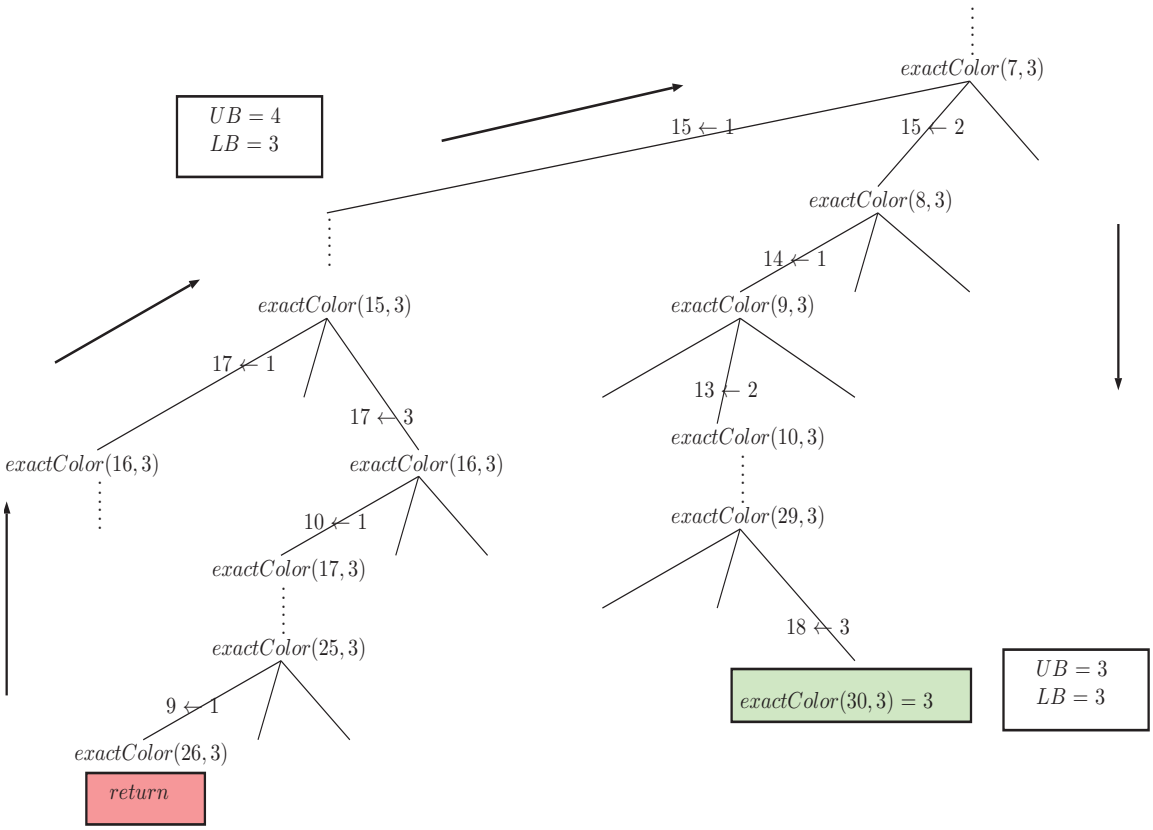

Figure 4.9: Backtracking and branching after getting an infeasible coloring

only color 1, 2 and 3 as options.

4.3 Column Selection and Tie-breaking Strategies

We already know the importance of selecting a column of highest saturation degree for coloring. It prevents unnecessary branching. If we select a column for which we have many choices to assign colors then the number branches will increase which affects the performance of the algorithm. So selecting the column with highest saturation degree is critical for the performance of the algorithm. Now, what happens if there are more than one columns with highest saturation degree? In this section, we will discuss the tie-breaking strategies when there are more than one columns with highest saturation degree.

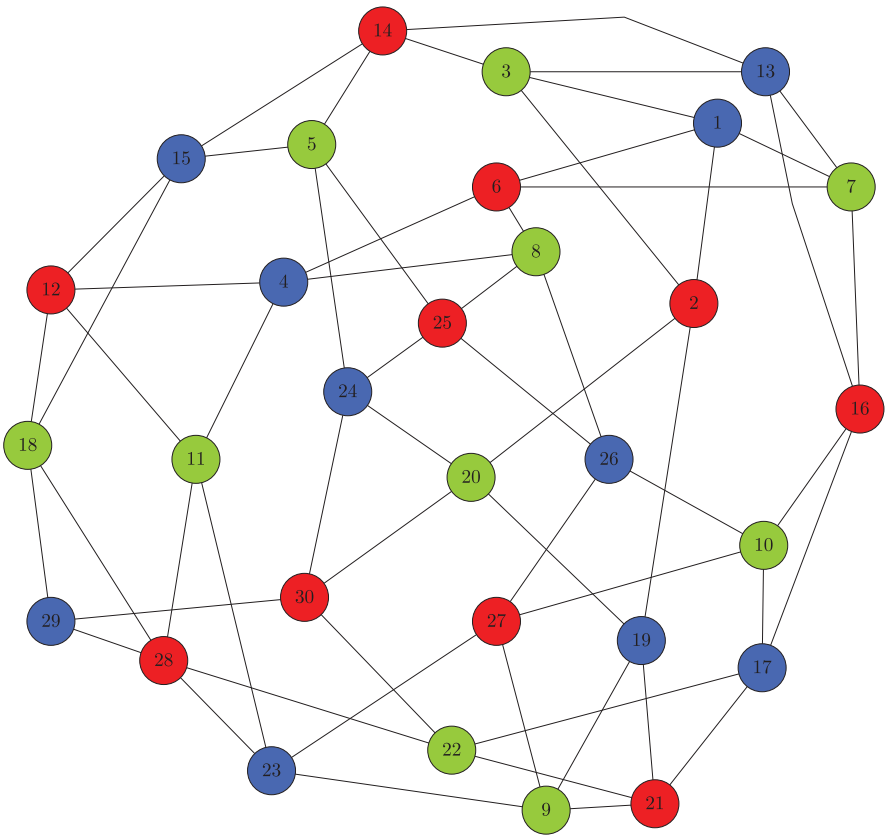

Figure 4.10: Optimal coloring of $G(B)$ using 3 colors

4.3.1 Simple Tie-breaking Strategy

The tie breaking strategy used in the example of Section 4.2 is very simple. We only select the column that is in the *HEAD* of highest saturation degree list. We also implement three more tie-breaking methods in DSJM. Two of them are from Segundo's [31] and Sewell's [32]. The third is a new tie-breaking rule.

4.3.2 Sewell's Rule

Sewell's rule to break tie is to select the vertex from the set of maximum saturation degree list which has the maximum number of common available colors in the neighborhood of uncolored vertices. This tie-breaking strategy reduces the number of subproblems because to break tie a vertex is selected that has the maximum number of common available colors in the neighborhood of uncolored vertices which reduces the number of available col-

ors for the uncolored vertices. The number of available colors reduction means a reduction in the number of subproblems. This tie-breaking rule is given in Figure 4.11

```
SELECTCOLUMNSEWELL()
1  max ← 0
2  column ← 0
3  for each vertex v ∈ maxSatDegree list
4       calculate U ← Feasible(u) ∩ Feasible(v)
5       where u ∈ N(v), u ≠ v and u not colored
6       if |U| > max
7            max ← |U|
8            column ← v
9  return column
```

Figure 4.11: Sewell's tie-breaking strategy

4.3.3 Segundo's PASS Rule

Segundo proposed another tie breaking strategy. This approach is computationally less expensive than Sewell's rule but has the same pruning effect. In Sewell's rule, the vertices in maximum saturation degree bucket look for common available colors among all of their uncolored neighbors. But according to Segundo's rule a vertex is selected similarly as Sewell, but while calculating the common available colors, it does not look all the uncolored neighbors instead it is restricted to the uncolored neighbors those are in the maximum saturation degree list. Segundo named this strategy PASS rule. Segundo's PASS rule is given in Figure 4.12

4.3.4 A New Tie-breaking Strategy

We propose a new tie-breaking rule. It is slightly different than Sewell's rule. The vertices in maximum saturation degree list look for common available colors among all of their uncolored neighbors of saturation degree at least 1. Our proposed tie-breaking rule is shown in Figure 4.13

```
SELECTCOLUMNPASS()
1  max ← 0
2  column ← 0
3  for each vertex v ∈ maxSatDegree list
4       calculate U ← Feasible(u) ∩ Feasible(v)
5       where u ∈ N(v), u ∈ maxSatDegre bucket, u ≠ v and u not colored
6       if |U| > max
7            max ← |U|
8            column ← v
9  return column
```

Figure 4.12: Segundo's tie-breaking strategy

```
SELECTCOLUMNNEW()
1  max ← 0
2  column ← 0
3  for each vertex v ∈ maxSatDegre bucket
4       calculate U ← Feasible(u) ∩ Feasible(v)
5       where u ∈ N(v), satDeg(u) ≥1, u ≠ v and u not colored
6       if |U| > max
7            max ← |U|
8            column ← v
9  return column
```

Figure 4.13: A new tie-breaking strategy

Sewell's rule helps to minimize subproblems by reducing available colors at deeper levels of the search tree. On the other hand, PASS rule reduces the number of available color to the vertices which already have the least number of available colors. We propose a new tie-breaking strategy that follows a middle ground. With this new tie-breaking strategy we try to achieve the optimal solution by exploring fewer subproblems with the same computational speed like others.

We implement all the above-mentioned tie-breaking strategies for selecting column when there are more than one columns with maximum saturation degree. The first one is simplest but requires more subproblems to get a feasible coloring. The Sewell's rule, PASS rule,

and our new implementation require fewer subproblems to get the same feasible coloring in most of the cases.

4.4 Numerical Experiments

In this section, we provide results from numerical experiments on some test instances. The data set for the experiments is obtained from The Matrix Market [1] and University of Florida Sparse Matrix Collection [10]. Healy and Ju [21] described a heap-based exact coloring algorithm but here we mainly compare our implementation with widely used Trick's implementation [27]. We do two types of experiments here. The purpose of the first experiment is to compare Trick's implementation of DSATUR based exact algorithm with our implementation. Here we call our implementation New Exact. In the second experiment, we display how the number of subproblems varies to get feasible colorings using different tie-breaking rules. As we have implemented all the tie breaking strategies in our implementation Trick's implementation is not consider in the second experiment. Let us discuss the first experiment first.

There are some differences between Trick's implementation and our implementation. For representing a graph, Trick used adjacency matrix, but we use efficient CSR and CSC data structures to associate column intersection graph with a matrix. In Trick's implementation n^2 space is needed to represent a graph. In our case it is $3nnz+m+n+2$. Suppose we are given a matrix of 130228 row and columns and 2032536 non-zero entries. Trick needs $130228 \times 130228 = 16959331984$ memory locations to represent the graphs. We need $3 \times 2032536 + 130228 + 130228 + 2 = 6358066$ memory locations to get the associated graph. For updating saturation degree Trick uses adjacency matrix as well. In our implementation, we use efficient bucket data structure for updating saturation degrees of the columns. Besides these, in our implementation, we do not search the maximal clique to determine the *LB* instead in our implementation ρ_{max} is the *LB*. These differences make our exact graph coloring algorithm more efficient than Trick's implementation. The description

of the test matrices is given in Table 4.1. The name of the matrices, the number of rows,

Table 4.1: Data set with lower bound

Name	***m***	***n***	***nnz***	**LB**	
				Trick	**New Exact**
bcsstk20	485	485	1810	11	11
bcsstm07	420	420	3836	26	26
dwt221	221	221	925	12	12
dwt878	878	878	4136	10	10
dwt918	918	918	4151	13	13
flower41	121	129	386	5	5
flower71	463	393	1178	5	5
flower81	625	513	1538	5	5
GL6D9	340	545	4349	17	28
gre512	512	512	2192	5	5
jagmesh1	936	936	3600	7	7
jagmesh5	1180	1180	4465	7	5
lnsp511	511	511	2796	11	11
lpireactor	318	808	2591	78	66
lunda	147	147	1298	21	21
mesh2e1	306	306	1162	10	10
mk9b1	378	36	756	4	2
n3c5b5	210	252	1260	6	6
n3c5b6	120	210	840	7	7
n4c5b10	120	630	1320	11	11
nos3	960	960	8402	18	18
nos5	468	468	2820	23	23
nos6	675	675	1965	5	5
plat1919	1919	1919	17159	19	19
poisson2D	367	367	2417	9	9
robot24c1mat5	404	302	15118	99	91
sherman1	1000	1000	2375	7	7
sphere3	258	258	1026	7	7
steam1	240	240	3762	21	21
west0655	655	655	2854	12	12

columns, and the number of non-zero entries are given. The *LB* we find in Trick's and in our implementation is also given in the table. To find the *LB* Trick uses an independent routine that searches for a maximal clique and terminates after a fixed number of iterations.

Although finding *LB* is straight forward in our implementation but in most of the cases *LB* we get from our implementation is same as Trick's. Only in 4 cases (jagmesh5, lpireactor, mk9b1, robot24c1mat5) our *LB* is smaller than Trick's and in one case (GL6D9) our *LB* is greater than Trick's. The test environment for the numerical experiments is same as described in Chapter 3, Section 3.3. We have divided the test results into three tables. The tables give us information about coloring, time and subproblems of Trick and New Exact implementations. All the clock time in the following tables are given in seconds. We see that New Exact is faster in most of the cases in terms of

- Number of subproblems explored per unit time
- Clock time, when the number of subproblems explored is identical.

In Table 4.2 we show the results of the test instances for which we get optimal coloring for both Trick and New Exact within the one-hour duration.

Table 4.2: Test results-1

Name	**Trick**			**New Exact**		
	Colors	**Time**	**Subproblems**	**Colors**	**Time**	**Subproblems**
bcsstm07	11	0.01	475	11	0	475
dwt221	12	0	210	12	0	210
dwt918	13	0.08	906	13	0.26	115457
flower41	5	0	4837	5	0	127
flower81	5	0.02	1249	5	0	554
gre512	5	0.52	110710	5	0.02	6414
jagmesh5	7	0.1	1174	7	0	1174
lpireactor	78	0.02	731	78	0	90
lunda	21	0	127	21	0	127
mesh2e1	10	0.01	297	10	0	297
mk9b1	7	0	1399	7	0	209
n4c5b10	11	0.05	897	11	0	886
nos3	18	0.2	943	18	0	943
nos6	5	0.03	671	5	0	671
robot24c1mat5	102	0.6	204	102	0.02	356
west0655	12	0.04	644	12	0	644

In Table 4.3 we show the results for the test instances for which New Exact gives optimal coloring within the one-hour duration, but Trick does not or vice versa. Only for one test case Trick gives optimal coloring but New Exact does not but for the rest of the test instances New Exact finds optimal coloring but Trick cannot. In this table the number of colors with asterisk (*) symbols mean we did not find any optimal coloring of the problems within the one-hour duration.

Table 4.3: Test results-2

Name	**Trick**			**New Exact**		
	Colors	**Time**	**Subproblems**	**Colors**	**Time**	**Subproblems**
dwt878	12*	-	5.29E+08	10	0	869
jagmesh1	9*	-	4.23E+08	8	10.04	6.17E+06
lnsp511	12*	-	8.23E+08	11	0	501
n3c5b6	7	0.6	412233	9*	-	3.56E+09
nos5	25*	-	9.11E+08	23	0.01	978
poisson2D	10*	-	1.18E+09	9	0	362
sherman1	8*	-	5.02E+08	7	0.03	1945

Table 4.4 displays the test results for which both Trick and New Exact cannot find optimal coloring within one hour. In the incomplete tests, we can find some interesting results as well. In some cases New Exact gets better coloring than Trick. If we look at the number of subproblems explored by Trick and New Exact, for most of the cases, we see that number of subproblems explored in New Exact is way more than Trick.

Table 4.4: Test results-3

Name	**Trick**		**New Exact**	
	Colors	**Subproblems**	**Colors**	**Subproblems**
bcsstm07	30*	1.02E+09	28*	1.14E+09
GL6D9	30*	7.72E+08	29*	3.52E+08
n3c5b5	10*	1.39E+09	10*	3.46E+09
plat1919	24*	2.24E+08	23*	9.82E+08
sphere3	9*	1.26E+09	9*	2.65E+09
steam1	23*	1.56E+09	22*	1.27E+09

In our second experiment, we take some matrices from Table 4.1 and show how the

number of subproblems varies to get feasible colorings for test instances when we use different tie-breaking mechanisms. We implemented four tie-breaking mechanisms described in Section 4.3 in New Exact. We only do not show the optimal/minimum coloring, and their corresponding required subproblems. We show all feasible coloring and the subproblems required to get the feasible coloring or the lowest coloring if we do not get optimal coloring within an hour in Table 4.5 . The number of subproblems is lesser than the simple tie-breaking strategy when we use other three tie-breaking strategies. In some cases, we do not get optimal coloring within an hour using simple tie-breaking strategy but get optimal coloring using the other(s). For any test problem if the entries of a row are filled with (-) it indicates we do not get any specific feasible coloring with that specific tie-breaking strategy. Suppose for bcsstm07, the first row of Segundo is empty. It means we do not get any coloring of 30 using Segundo but we get coloring of 30 using other tie-breaking strategies. If the smallest feasible color has (*) sign with it, it indicates we do not get optimal coloring within the one-hour duration.

Table 4.5: Comparison between different tie-breaking strategies of New Exact

Name	Simple			Sewell			Segundo			New		
	Colors	**Time**	**Subproblems**	**Colors**	**Time**	**Subproblems**	**Colors**	**Time**	**Subproblems**	**Colors**	**Time**	**Subproblems**
bcsstm07	30	0	395	30	0	395	-	-	-	-	-	-
	29	0.03	16105	29*	5.7	7.11E+05	29	0	395	29*	0.01	395
	28*	57.61	1.62E+07	-	-	-	28*	0	481	-	-	-
dwt878	10	0	869	10	0	869	10	0	869	10	0	869
flower41	5	0	161	5	0	172	5	0	282	5	0	127
flower71	5	0	439	5	0	402	5	0	485	5	0	400
jagmesh1	9	0	983	9*	0.04	932	9*	0.02	968	9*	0.04	963
	8	10.04	6.17E+06	-	-	-	-	-	-	-	-	-
jagmesh5	7	0	1174	7	0.11	1174	7	0.02	1174	7	0.12	1174
lnsp511	12	0	501	12	0	501	-	-	-	12	0	501
	11	0	1005	11	0	920	11	0	501	11	0	595
lunda	-	-	-	-	-	-	22	0	127	-	-	-
	21	0	127	21	0	127	21	0	243	21	0	127
mk9b1	7	0	414	7	0	209	7	0	197	7	0	209
n3c5b5	12	0	247	12	0	247	-	-	-	12	0	247
	11	0	301	11	0	289	11	0	247	11	0	344
	10*	14.22	1.36E+07	10*	7.11	1.59E+06	10*	0	2104	10*	2.74	667379
n3c5b6	-	-	-	-	-	-	-	-	-	13	0	204
	-	-	-	12	0	204	12	0	204	12	0	245
	11	0	204	11	0	253	11	0	1347	11	0	323
	10	0	6738	10	1.55	398017	10	0.17	69813	10	3.12	809260
	9*	1225.99	1.22E+09	9*	432.45	9.88E+07	9*	3217.28	1.34E+09	9*	16.28	3.74E+06
nos3	20	0	943	-	-	-	-	-	-	-	-	-
	19	0.01	2406	19	0.01	943	-	-	-	-	-	-
	18	0.01	3266	18	0.2	1844	18	0	943	18	0.01	943
nos5	24*	6.2	2.37E+06	24*	0	446	24*	0	446	24*	0.01	446
	-	-	-	-	-	-	23	0.01	978	-	-	-
plat1919	24	0.01	1901	-	-	-	24	0.01	1901	24*	0.03	1901
	23*	0.02	3359	-	-	-	23*	0.03	3871	-	-	-
sphere3	-	-	-	11	0	338	11	0	252	-	-	-
	10	0	252	10	0.06	23568	10	0	1197	10	0	252
	9*	0.88	702048	9*	0.6	176461	9*	6.6	2.69E+06	9*	9.49	2.46E+06
steam1	-	-	-	24*	0.03	7813	24*	0	220	24	0	0.01
	23	0	220	-	-	-	-	-	-	23*	0.01	537
	22*	4.16	1.91E+06	-	-	-	-	-	-	-	-	-

Chapter 5

Conclusion and Future Works

In this thesis, we have provided a detailed study on the efficient data structures of DSJM software. The heuristic ordering and partitioning algorithms used in DSJM give the best coloring with quickest possible time compared to other existing implementations. We show the efficiency of DSJM and the advantages of using cache-friendly data structures using several numerical experiments and by comparing DSJM with another existing software toolkit. There are a lot of scopes to extend DSJM toolkit, but without the proper understanding of the data structures and implementations, it would be difficult for anyone. Step by step examples of how buckets change in the ordering and partitioning algorithms can be a useful material to study for future researchers.

With the help of a clear understanding of the implementation of DSJM, we extended DSJM by implementing an exact graph partitioning algorithm. The exact partitioning algorithm can find optimal coloring for small instances quite quickly, and we have shown that in many cases where the other existing implementation cannot find optimal coloring but our implementation can provide optimal coloring within a reasonable amount of time. We also implemented four tie-breaking strategies for selecting a column in every step and showed selecting a column strategically can reduce the number of subproblems in getting feasible solutions. The exact coloring algorithms still do not find optimal coloring in a reasonable amount of time for large test instances. We are trying to find a solution. A combined method proposed by Hossain et al. [24] uses exact coloring algorithm to color a critical submatrix of a large matrix. This combined method does not guarantee to give optimal coloring but

gives promising coloring than the heuristic approaches. Our exact coloring implementation is incorporated in the combined coloring algorithm.

5.1 Future Works

At present the column partitioning in DSJM is done using one-sided compression (row compression). It can be extended to enable two-sided compression.

Heuristic approaches work very fast for large instances, but optimal partitioning is not guaranteed. On the other hand, the exact approach provides optimal coloring but is not practical for large instances. Combined approach can be a good direction for finding optimal coloring for large instances. At present, the combined approach does not guarantee optimal coloring. Further research can be done on combined approach.

The real world problems are getting larger every day. To deal with the very large instances, parallel implementation on shared memory multi-processor system is an interesting research direction. We gave a try to implement some ordering and partitioning algorithms of DJSM using OpenMP (Open Multi-Processing) Application Programming Interface (API) [8]. We found it difficult to parallelize graph operations using existing data structures. Further study on parallel implementation can be a good and exciting research direction.

Bibliography

[1] The Matrix Market Collection. http://math.nist.gov/MatrixMarket/matrices.html. Accessed: 2017-01-13.

[2] Power network patterns. http://math.nist.gov/MatrixMarket/data/Harwell-Boeing/bcspwr/bcspwr01_lg.html. Accessed: 2017-03-03.

[3] D. Brélaz. New Methods to Color the Vertices of a Graph. *Commun. ACM*, 22(4):251–256, April 1979.

[4] J. R. Brown. Chromatic Scheduling and the Chromatic Number Problem. *Management Science*, 19(4-part-1):456–463, 1972.

[5] R. E. Bryant and D. R. O'Hallaron. *Computer Systems: A Programmer's Perspective*. Addison-Wesley Publishing Company, USA, 2nd edition, 2010.

[6] T. F. Coleman and J. J. Moré. Estimation of Sparse Jacobian Matrices and Graph Coloring Problems. *SIAM Journal on Numerical Analysis*, 20(1):187–209, 1983.

[7] A. R. Curtis, M. J. D. Powell, and J. K. Reid. On the Estimation of Sparse Jacobian Matrices. *IMA Journal of Applied Mathematics*, 13(1):117, 1974.

[8] L. Dagum and R. Menon. OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55, January 1998.

[9] T. A. Davis. *Direct Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, 2006.

[10] T. A. Davis and Y. Hu. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, December 2011.

[11] E. D. Dolan and J. J. Moré. Benchmarking optimization software with performance profiles. *Mathematical Programming*, 91(2):201–213, 2002.

[12] I. S. Duff, R. G. Grimes, and J. G. Lewis. Sparse Matrix Test Problems. *ACM Trans. Math. Softw.*, 15(1):1–14, March 1989.

[13] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *40th Annual Symposium on Foundations of Computer Science*, pages 285–297, 1999.

[14] G. Gallo and S. Pallottino. Shortest path algorithms. *Annals of Operations Research*, 13(1):1–79, Dec 1988.

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

[16] A. H. Gebremedhin, F. Manne, and A. Pothen. What Color Is Your Jacobian? Graph Coloring for Computing Derivatives. *SIAM Rev.*, 47(4):629–705, April 2005.

[17] A. H. Gebremedhin, D. Nguyen, M. M. A. Patwary, and A. Pothen. ColPack: Software for Graph Coloring and Related Problems in Scientific Computing. *ACM Trans. Math. Softw.*, 40(1):1:1–1:31, October 2013.

[18] 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.

[19] G. H. Golub and C. F. Van Loan. *Matrix Computations (3rd Ed.)*. Johns Hopkins University Press, Baltimore, MD, USA, 1996.

[20] M. Hasan, S. Hossain, A. I. Khan, N. H. Mithila, and A. H. Suny. *DSJM: A Software Toolkit for Direct Determination of Sparse Jacobian Matrices*, pages 275–283. Springer International Publishing, Cham, 2016.

[21] P. Healy and A. Ju. *An Experimental Analysis of Vertex Coloring Algorithms on Sparse Random Graphs*, pages 174–186. Springer International Publishing, Cham, 2014.

[22] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011.

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

[24] S. Hossain and A. H. Suny. Determination of Large Sparse Derivative Matrices: Structural Orthogonality and Structural Degeneracy. In *Proceedings of 15th Cologne-Twente workshop on Graphs and Combinatorial Optimization 2017*, pages 83–87, Cologne, Germany, 2017.

[25] F. T. Leighton. A graph coloring algorithm for large scheduling problems. *Journal of research of the national bureau of standards*, 84(6):489–506, 1979.

[26] E. Malaguti and P. Toth. A survey on vertex coloring problems. *International Transactions in Operational Research*, 17(1):1–34, 2010.

[27] A. Mehrotra and M. A. Trick. A Column Generation Approach For Graph Coloring. *INFORMS Journal on Computing*, 8:344–354, 1995.

[28] N. Nethercote and J. Seward. Valgrind: A Program Supervision Framework. *Electronic Notes in Theoretical Computer Science*, 89(2):44 – 66, 2003. RV '2003, Runtime Verification (Satellite Workshop of CAV '03).

[29] J. Park, M. Penner, and V. K. Prasanna. Optimizing Graph Algorithms for Improved Cache Performance. *IEEE Trans. Parallel Distrib. Syst.*, 15(9):769–782, September 2004.

[30] J. Peemöller. A Correction to Brelaz's Modification of Brown's Coloring Algorithm. *Commun. ACM*, 26(8):595–597, August 1983.

[31] P. S. Segundo. A new DSATUR-based algorithm for exact vertex coloring. *Computers & Operations Research*, 39(7):1724 – 1733, 2012.

[32] E. C. Sewell. An improved algorithm for exact graph coloring. *DIMACS series in discrete mathematics and theoretical computer science*, 26:359–373, 1996.

[33] W. Tristram and K. Bradshaw. Performance Optimisation of Sequential Programs on Multi-core Processors. In *Proceedings of the South African Institute for Computer Scientists and Information Technologists Conference*, SAICSIT '12, pages 119–128, New York, USA, 2012.