DSJM: A Software Toolkit for Direct Determination of Sparse Jacobian Matrices

Mahmudul Hasan
Bachelor of Science, Islamic University of Technology, Bangladesh, 2006

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

DSJM: A SOFTWARE TOOLKIT FOR DIRECT DETERMINATION OF SPARSE JACOBIAN MATRICES

MAHMUDUL HASAN

Approved:

Signature *Date*

Supervisor:

Committee Member:

Committee Member:

Chair, Thesis Examination Committee:

I dedicate this thesis to my parents.

Abstract

DSJM is a software toolkit written in portable C++ that enables direct determination of sparse Jacobian matrices whose sparsity pattern is a priori known. Using the seed matrix $S \in R^{n \times p}$, the Jacobian $A \in R^{m \times n}$ can be determined by solving $AS = B$, where $B \in R^{m \times p}$ has been obtained via finite difference approximation or forward automatic differentiation. Seed matrix S is defined by the nonzero unknowns in A. DSJM includes well-known as well as new column ordering heuristics. Numerical testing is highly promising both in terms of running time and the number of matrix-vector products needed to determine A.

Acknowledgments

I express my deep acknowledgment and gratitude towards my M.Sc. supervisor Dr. Shahadat Hossain. Without his guidance and effort this thesis would have never been complete. Contents of this thesis is based on the joint research work I have done with Dr. Hossain.

I also express my gratitude towards my M.Sc. supervisory committee members Dr. Daya Gaur, and Dr. Saurya Das for their valuable advice and guidance. I am thankful to Dr. Trond Steihaug, for his feedback and ideas on the research work.

I am grateful to Dr. Hossain, and Graduate School of University of Lethbridge for providing me the financial resources needed to support the research work.

I thank my friends, colleagues and families for their support and forbearance.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

An important computational step in many numerical algorithms solving complex scientific and engineering problems is to compute or estimate the first or higher-order derivatives of a vector function of several independent variables [11, 26]. Complex real-world phenomena e.g., atmospheric dynamics are usually studied by building models (differential equations) for constituent natural processes. The numerical procedures in those models often require the solution of systems of nonlinear equations or minimization of some nonlinear function of large number of variables. A frequently used algorithm to solve these problem is some variant of Newton's method.

Algorithm 1: Newton's method for solving system of nonlinear equations

input: For an initial approximation $\mathbf{x} \in \mathbf{R}^n$

1 **for** $j \leftarrow 0$ **to** *convergence* **do**
2 | Evaluate $b = F(\mathbf{x})$;
3 | Determine $J = F'(\mathbf{x})$;
4 | Solve for $J\mathbf{s} = -b$;
5 | $x \leftarrow \mathbf{x} + \mathbf{s}$;
6 **end**

Figure 1.1: Model Newton's Algorithm

Newton's method finds a solution of a system of nonlinear equations specified by

$$F(\mathbf{x}) = 0$$

where $F(\mathbf{x}) : \mathbf{R}^n \mapsto \mathbf{R}^m$ is a vector valued function on $\mathbf{x}$. Starting from an initial approximation, newton's method improves the solution iteratively.

Each iteration requires one evaluation of $F(\mathbf{x})$ and its derivative $F'(\mathbf{x})$ at a given point

x. So, in a large number of scientific and engineering problems determination of derivatives or Jacobian of $F(\mathbf{x})$ is a necessary computational step.

In most of the cases we can only approximate the value of the Jacobian $F'(\mathbf{x})$ using numerical methods, most notably finite differencing or automatic differentiation [19]. Scientific and engineering problems often produces large Jacobian matrices which are sparse, or has structural patterns in them. Though there has been a significant improvement in the algorithmic methods for determining the Jacobian of a function, there is a gap between theory and implementation. In our thesis, we present a software toolkit which tries to fill the gap by providing a tool to determine large Jacobian matrices efficiently by exploiting the sparsity frequently found in the real-world problems. Though important, the implementation does not try to use the known specific structural pattern of the sparsity (i.e. tri-diagonal matrix, banded matrix).

Given a nonlinear vector function

$$F(\mathbf{x}) = [\, f_1(\mathbf{x}) \quad f_2(\mathbf{x}) \quad \dots \quad f_m(\mathbf{x})]^T, \qquad \mathbf{x} \in \mathbf{R}^n \tag{1.1}$$

we want to compute the Jacobian matrix $F'(\mathbf{x})$ at a given **x**, where $F'(\mathbf{x})$ is given by

$$F'(\mathbf{x}) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \dots & \frac{\partial f_1}{\partial x_i} & \dots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \dots & \frac{\partial f_2}{\partial x_i} & \dots & \frac{\partial f_2}{\partial x_n} \\ \frac{\partial f_m}{\partial x_1} & \frac{\partial f_m}{\partial x_2} & \dots & \frac{\partial f_m}{\partial x_i} & \dots & \frac{\partial f_m}{\partial x_n} \end{bmatrix} \tag{1.2}$$

Finite differencing can approximate n columns of Jacobian matrix with $n+1$ function evaluations. In the forward difference formula

$$\frac{\partial f}{\partial x_i}(\mathbf{a}) \approx \frac{1}{\varepsilon}[F(\mathbf{a} + \varepsilon \mathbf{e_i}) - F(\mathbf{a})], \tag{1.3}$$

one needs to evaluate F at $\mathbf{a}$ and n neighboring points $(\mathbf{a}+\varepsilon\mathbf{e_i})$ where, $i = 1,2,\ldots,n$, and $\varepsilon > 0$ is a small interval and $\mathbf{e_i}$ is i^{th} co-ordinate vector.

$$A = \begin{pmatrix} 0 & \times & \cdots & 0 & 0 & 0 \\ \times & \times & \cdots & 0 & 0 & 0 \\ \times & 0 & \cdots & 0 & \times & 0 \\ \vdots & \vdots & \cdots & & \vdots & \vdots \\ \times & 0 & \cdots & 0 & \times & \times \\ 0 & \times & \cdots & 0 & 0 & \times \\ & j & & & k & \end{pmatrix}$$

Figure 1.2: Structure of a Jacobian Matrix

Definition Two columns are called *structurally orthogonal*, if they do not have nonzero entries in same row.

In Figure 1.2, columns j and k are structurally orthogonal as they do not have overlapping nonzero entries in the same row position. Curtis, Powell and Reid [9] showed that if two columns j and k are structurally orthogonal ,these two columns can be approximated in a single evaluation instead of two, noting that

$$A_j + A_k \approx \frac{1}{\varepsilon}[F(\mathbf{a}+\varepsilon(\mathbf{e_j}+\mathbf{e_k})) - F(\mathbf{a})]. \tag{1.4}$$

They proposed that by grouping the n columns into p structurally orthogonal groups , the number of function evolutions required to compute large sparse Jacobian matrices can be reduced significantly, thus introducing column partitioning problem as a kernel operation in determining Jacobians efficiently.

Further analysis on Jacobian determination problem is given in [16], [22] and [21]. Coleman and Moré [8] first showed that this problem can be modeled also as a graph coloring problem. Considering each column as a vertex in an intersection graph (see Section

2.4.1), Coleman and Moré developed efficient heuristics for column partitioning. Further development on this idea was carried out in [21] by constructing CSegGraph [21].

We assume that the sparsity pattern of the Jacobian matrix is known a priori and is independent of the actual values of $\mathbf{x}$, or can be computed as in Automatic Differentiation [19]. We also assume that for one or more components of $F(\mathbf{x})$ we need to compute the whole vector $F(\mathbf{x})$. It is more efficient to evaluate vector F than to evaluate each component of F separately, as

- common sub-expressions are evaluated only once, and
- F might be a computer subroutine that returns the vector F evaluated at $\mathbf{x}$ whose code is not available directly to users.

DSM [7] was the only software since its release in 1983 for determining Jacobian matrices. DSM is very efficient and works well on the target problem. But as DSM was programmed in FORTRAN(F77) it cannot take advantage of dynamic memories, and other modern development in programming languages e.g. object orientation. *Colpack* [17] and *DSJM* are two softwares that address the same problem using modern implementation. *Colpack* models column partitioning problem as bipartite graph, wherein *DSJM* builds sparse matrix primitives where graph theoretic techniques are implemented using efficient sparse data structures.

1.1 Contribution

DSJM implements proven coloring and ordering heuristics, as well as some novel ones. It also makes available a heuristic coloring technique with no known alternative implementation, which proved to be of more effective than the other known heuristics. Graph algorithms typically display relatively small floating-point operations count per memory access

resulting in degraded performance on traditional hierarchical-memory computer systems. Our implementation of the ordering and coloring algorithms, with the help of efficient sparse data structures, allow the kernel operations to be performed in a cache-friendly way to minimize *cache misses* due to irregular data access. Choice of C++ as implementation language equips DSJM with dynamic and efficient memory management, as well as wider scope for extensibility and ease of use through object oriented design. Along with making available itself as a linkable C++ library, the routines can also be used from MATLAB tools. DSJM was successfully interfaced with MAD (Matlab Automatic Differentiation) [13]. Part of this work has been published as an extended abstract in SIAM workshop on Combinatorial Scientific Computing, 2009. It was also presented as a talk in IBM Cascon Conference, 2010, and University of Lethbridge Optimization Seminar, as well as a poster in CORS/MITACS conference, 2010.

Chapter 2

Background and Sparse Matrix Data Structure

In this chapter we review some preliminary graph theoretic definitions necessary for this thesis, and introduce sparse matrix data structure for representing graphs.

2.1 Graph

A graph G is a pair (V,E), where V is a finite set of vertices and E is a binary relation over V. Each element in E is called an edge and is a set $\{u,v\}$ such that $u,v \in V$.

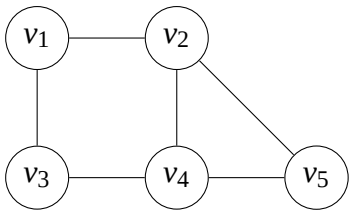

Figure 2.1: A Graph $G = (V,E)$, where $V = \{v_1, v_2, v_3, v_4, v_5\}$ and edge set $E = \{\{v_1, v_2\}, \{v_1, v_3\}, \{v_2, v_4\}, \{v_3, v_4\}, \{v_4, v_5\}\}$. The vertices are shown as circles and edges as lines connecting two vertices.

We call two vertices u and v *adjacent* if and only if $\{u,v\} \in E$. The *neighborhood*, $N(v)$, of a vertex v is defined as the set of all vertices $u \neq v$, such that $\{u,v\} \in E$. The *degree* of a vertex v is defined as $d(v) = |N(v)|$.

The graph induced by $V' \subset V$, denoted $G[V'] = (V',E')$, is the subgraph of $G = (V,E)$ where $E' = \{\{u,v\} \in E | u,v \in V'\}$.

2.2 Seed matrix computation

In this thesis we consider the problem of finding minimum cardinality structurally orthogonal column partitioning of a Jacobian matrix. The main goal of this thesis is the design and

implementation of efficient data structure and partitioning related algorithms. As we have seen in Chapter 1, the nonzero elements of a Jacobian matrix, **A**, can be obtained using an identity matrix as a trivial seed matrix S, where $S = I_n$, via finite difference formula,

$$\frac{\partial f}{\partial x_i}(\mathbf{x}) \approx \frac{1}{\varepsilon}[F(\mathbf{x}+\varepsilon\mathbf{e_i}) - F(\mathbf{x})].$$

The determination of a sparse Jacobian matrix can also be viewed as a computation of p matrix-vector products $AS = B$:

$$\left.\frac{\partial F(x+ts)}{\partial t}\right|_{t=0} = F'(x)s \approx As = \frac{1}{\varepsilon}[F(x+\varepsilon s) - F(x)] \equiv b. \tag{2.1}$$

Nonzero elements in A can be recovered by solving $AS = B$ using a direct or indirect method.

Definition A reduced seed matrix $\hat{S}_i \in R^{\rho_i \times p}$ is defined as

$$\hat{S}_i = S(\mathcal{J}_i, :),$$

where $\mathcal{J}_i$ denotes a vector containing the column indices of the nonzero entries in row i of A.

We say that A is determined directly, if S satisfies the property that each reduced seed matrix $\hat{\mathbf{S}}_\mathbf{i}$ has a $\rho_i \times \rho_i$ submatrix that is a permuted diagonal matrix.

Structurally orthogonal column partitioning gives us a seed matrix $S \in \{0,1\}^{n\times p}$ which follows the properties for direct determination, where

$$S(:,l) = \sum_{j\in C_l} e_j, \qquad l = 1,2,\ldots,p$$

and C_l is a set of column indices that are structurally orthogonal. Hence, structurally or-

thogonal column partitioning problem can also be reformulated as a seed matrix computation.

It has been observed in [8] that the seed matrix computation problem can be formulated as the vertex coloring of an associated graph, $G(A)$.

2.3 Intractability

Polynomial-time algorithms have worst-case running time of $O(n^k)$ for an input size n, and a constant k. Polynomial-time algorithms are generally considered tractable [15]. We define the class of *polynomial-time solvable* problems that allows *polynomial-time* algorithms for solving them. Similarly, the class of *polynomial-time verifiable* problems are defined as those set of problems which, given an answer, allows to be verified whether the answer corresponds to the solution or not. The class of *polynomial-time solvable* problems is denoted by P, and the class of *polynomial-time verifiable* problems is denoted by NP. By this definition, $P \subset NP$.

If there is a *polynomial-time* algorithm which converts any input of problem B to an equivalent input of problem A in such a way that the solution computed by an algorithm to solve A, is also a solution to problem B and vice versa, then the conversion algorithm is called a *polynomial-time reduction*. A problem in NP to which all other problems in NP can be polynomially-reduced is called an *NP-Complete* problem.

As any algorithm to solve an *NP-Complete* problem can solve all other problems in NP, the class of *NP-Complete* problems are considered to be the hardest in NP. Moreover, no *polynomial-time* algorithm has been found yet to solve an *NP-Complete* problem. It is generally safe to say that there is no polynomial time algorithm for an *NP-Complete* problem unless $P = NP$.

2.4 Graph Coloring

Given a graph $G = (V,E)$, a p-coloring is a function $\phi : V \to \{1,\ldots,p\}$ such that $\phi(u) \neq \phi(v)$ if $\{u,v\} \in E$. The minimum value for p is called the *chromatic number* $\chi(G)$ of graph G. It has been shown that given an arbitrary graph G, to decide whether or not it has a *p-coloring* is *NP-Complete* [15]. Since this *p-coloring* problem is *NP-Complete*, finding the value of minimum p cannot be any easier than the decision version of the problem.

2.4.1 Intersection Graph

Given a matrix A, we can construct a graph $G(A) = (V,E)$ in a way such that each $v_i \in V$ corresponds to a unique column i, $i = 1,2,\ldots,n$, in A. We define the edge $\{v_i,v_j\} \in E$ if and only if columns i and j share at least one nonzero in some row. $G(A)$ is called the *intersection graph* of A. It has been shown that coloring of the intersection graph $G(A)$ induces a structurally orthogonal column partition in matrix A and vice versa [8].

2.5 Heuristics

Practical way of looking into this *NP-Hard* problem is to utilize heuristics. Heuristic algorithms for graph coloring can be broadly categorized into greedy constructive algorithms and meta-heuristic methods [24]. Meta-heuristic methods include local search algorithms, tabu search, simulated annealing, genetic and evolutionary algorithms and etc. Although there are different types of heuristics, in this thesis we discuss greedy constructive heuristics only, because majority of the heuristics are not practically applicable for Jacobian determination due to the large size of the input matrix [24].

The simplest greedy constructive algorithms is the greedy sequential algorithm (SEQ), where each vertex v_i is assigned the lowest indexed color class which contains no vertices

adjacent to v_i. Carefully pre-ordered sequence of vertices to the SEQ algorithm can achieve better coloring [8].

In *Largest First Ordering* (LFO), the vertices $V = \{v_1, v_2, \ldots, v_n\}$ are sorted in non-decreasing degrees in G, and then the ordering is provided to the SEQ method.

Assuming that the vertices $V' = \{v_n, v_{n-1}, \ldots, v_{i+1}\}$ have already been ordered, the i-th vertex in *Smallest Last Ordering* (SLO) is an unordered vertex u such that $deg(u)$ is minimum in $G[V \setminus V']$.

After ordering the vertices $V' = \{v_1, v_2, \ldots, v_{i-1}\}$ the i-th vertex in *Incidence Degree Ordering* (IDO) [8] is an unordered vertex $u \in V \setminus V'$ such that $deg(u)$ is maximum in $G[V']$. Ties are broken by choosing the vertex that has largest degree in G .

Definition The chromatic degree kdeg(u) of a vertex u is defined as the number of unique color(s) present in the neighborhood of u.

In *Saturation Degree Ordering* (SDO) [5] we order and color first before choosing the next vertex; Assume that the vertices $V' = \{v_1, v_2, \ldots, v_{i-1}\}$ have been ordered and colored, the i-th vertex in this order is an unordered vertex u such that $kdeg(u)$ is largest in $G[V']$. Ties are broken by choosing the vertex that has the largest degree in $G[V \setminus V']$.

Recursive-largest-first (RLF) [23] algorithm partitions the vertex set V into $V_1, V_2, \ldots, V_p$ independent sets, and constructs p color classes. The first vertex of V_i is chosen in a way such that it has the largest degree in $G[V \setminus \bigcup_{j=1}^{i-1} V_j]$ induced graph, and adjacent vertices of v_1 are added to the inadmissible set U. RLF continues adding vertices to the independent set V_i, by choosing v_k which has the largest number of adjacent vertices in the set U at k-th step, and neighbors of v_k are also added to U.

2.6 Data Structure

In this section we are going to describe the data structure which stores sparse matrices in computer memory for our heuristic algorithms. We wanted to use a data structure which can exploit the sparsity, and simultaneously can store the intersection graph implicitly. *Compressed Column Storage* and *Compressed Row Storage* schemes appear to be a suitable fit for our purpose and they form the backbone of our implementation. In the following section, we describe the *Compressed Column Storage* and *Compressed Row Storage* schemes.

2.6.1 Compressed Column Storage (CCS)

The Compressed Column Storage (CCS) puts the row indices of nonzero elements of subsequent columns in an integer array (`row_ind`). Nonzero elements are stored in the same order in a floating point array (`val`). An integer array (`col_ptr`) is created to store the beginning indices of the columns in `row_ind`. Assuming that we have a matrix $A^{m\times n}$ with nnz number of nonzero elements, we need $2nnz+n+1$ number of memory locations instead of n^2 to store A.

Row indices of nonzero elements in column j are found to be as `row_ind[col_ptr[j]]` to `row_ind[col_ptr[j+1]-1]`.

2.6.2 Compressed Row Storage (CRS)

Analogous to Compressed Column Storage, Compressed Row Storage (CRS) puts column indices of the nonzero elements of subsequent rows in an integer array, `col_ind`. Integer array `row_ptr` is created as a pointer to the column indices. In our implementation we do not store nonzero elements in *CRS* scheme to save space. Thus CRS requires $nnz+m+1$ number of memory locations.

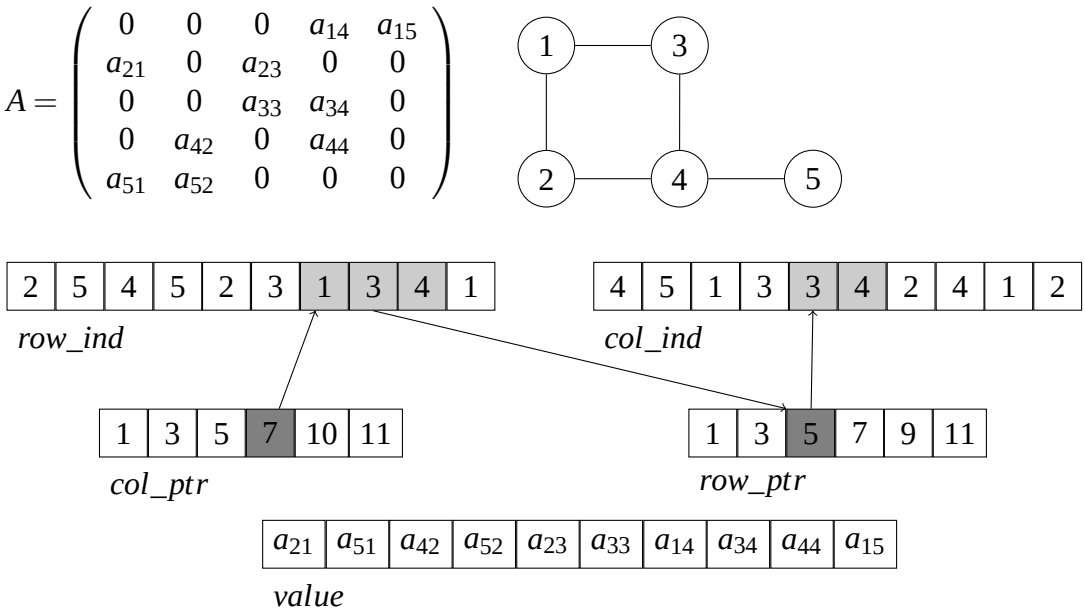

Figure 2.2: Compressed Column and Compressed Row data structure for the sparse matrix *A*. The intersection graph of the matrix *A* is shown on the right. Integer array, `row_ind`, stores the row indices of the nonzero elements of subsequent columns. Nonzero elements are stored in the same order in the floating point array `value`. Integer array `col_ptr` stores the beginning memory location for columns in `row_ptr`. For example, shaded cells in `col_ptr` and `row_ind` shows the corresponding entries for column 4. These three arrays, `value`, `row_ind` and `col_ptr` forms the Compressed Column Storage format. `Col_ind` and `row_ptr` stores the Compressed Row Storage format, and corresponding entries for row 3 are shown as shaded.

Column indices of nonzero elements in row *i* are found to be as `col_ind[row_ptr[i]]` to `col_ind[row_ptr[i+1]-1]`.

In our implementation we use both *CCS* and *CRS*, resulting in total of $3nnz + m + n + 2$ number of memory locations used for sparse matrix storage.

Other important data structures will be introduced and described in Chapter 3.

Chapter 3

Efficient Implementation of Ordering and Graph Coloring

In this chapter we describe implementation details for different column ordering strategies and asymptotic complexity of the algorithms and data structures. We also describe detail illustrations of the algorithms with *CRCS* and other supporting data structures.

The column intersection graph need not be constructed explicitly if we use *CRCS* data structure for matrices. *CRCS* allows us to compute the neighborhood of a column j efficiently by providing both row-oriented and column-oriented sparsity pattern. For example, for a given column j, we can compute the neighborhood using the algorithm described in Figure 3.1. Figure 3.2 illustrates a matrix A, its corresponding intersection graph $G(A)$, and its *CRCS* representation.

Algorithm 2: Neighborhood Computation using CRCS

input: column $jcol$

```
1 for jp ← jpntr[jcol] to jpntr[jcol+1] − 1 do
2 |   ir ← row_ind[jp] ;
3 |   for ip ← ipntr[ir] to ipntr[ir+1] − 1 do
4 |   |   ic ← col_ind[ip] ;
5 |   |   // ic is a neighbor to j.
6 |   end
7 end
```

Figure 3.1: Neighborhood computation using CRCS

3.1 Constructive Greedy Coloring

Greedy coloring algorithm can be considered as the *de facto* constructive greedy heuristics for column partitioning. DSM [7] and ColPack [17] use greedy coloring for column partitioning problem as well. An algorithm for greedy coloring is given in Figure 3.4. The

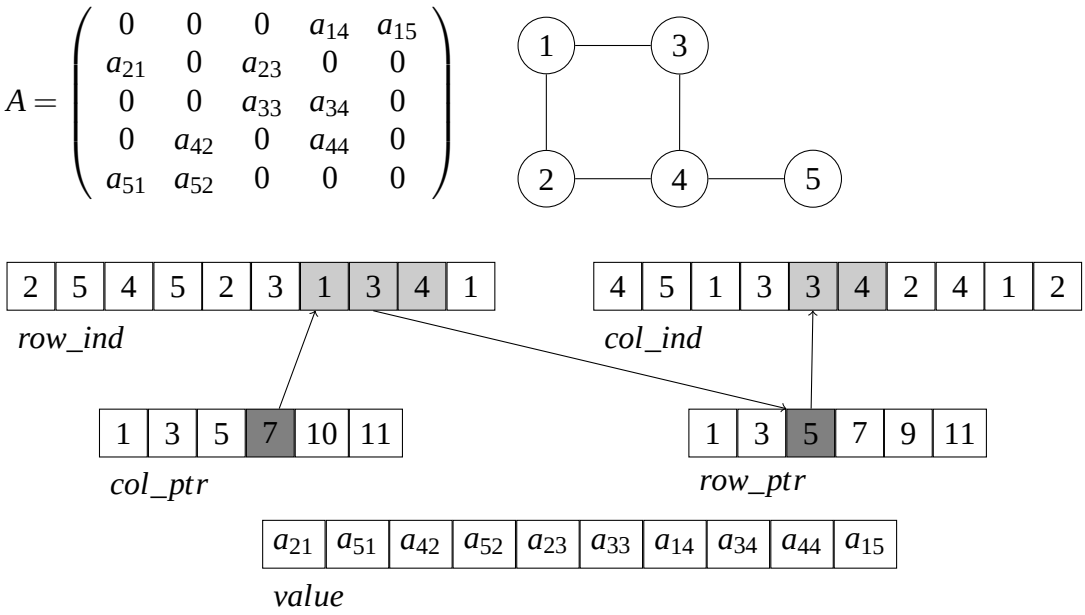

Figure 3.2: Compressed Column and Compressed Row data structure for the sparse matrix *A*. The intersection graph of the matrix *A* is shown on the right. Integer array, `row_ind`, stores the row indices of the nonzero elements of subsequent columns. Nonzero elements are stored in the same order in the floating point array `value`. Integer array `col_ptr` stores the beginning memory location for columns in `row_ptr`. For example, shaded cells in `col_ptr` and `row_ind` shows the corresponding entries for column 4. These three arrays, `value`, `row_ind` and `col_ptr` forms the Compressed Column Storage format. `Col_ind` and `row_ptr` stores the Compressed Row Storage format, and corresponding entries for row 3 are shown as shaded.

algorithm is divided into the following major computational steps:

1. Initialization, lines $1-4$.

2. Neighborhood Computation, lines $7-10$

3. Tagging, line 11

4. Coloring, lines $15-25$

Initialization Lines $1-4$ from the algorithm initializes necessary data structures, *color* and *tag*.;
Neighborhood Computation Given a column *jcol*, lines $7-13$ uses *CRCS* data structure to find all the neighbors for column *jcol*.
Tagging We tag the color of a neighbor *ic* with the value of *jcol* as soon as we compute *ic* in line 11.
Coloring We find the minimum color for the current column *jcol* from lines $15-20$, by taking the first color which has not been assigned to any neighbor of column *jcol*.
Neighborhood computation, tagging and coloring is done for each column *jcol*, which is taken from the given ordering *order* in the loop which covers from line $6-26$.

Figure 3.3: Major computational steps for sequential coloring

Algorithm 3: Sequential Coloring Algorithm

input : *order*, an integer array of size *n*, containing a permutation of $\{1 \ldots n\}$
output: *color*, an integer array of size *n*

```
1  for j ← 1 to n do
2  |   color[j] ← n ;
3  |   tag[j] ← n ;
4  end
5  maxgrp ← 0;
6  for seq ← 1 to n do
7  |   jcol ← order[seq] ;
8  |   for jp ← jpntr[jcol] to jpntr[jcol+1] − 1 do
9  |   |   ir ← row_ind[jp] ;
10 |   |   for ip ← ipntr[ir] to ipntr[ir+1] − 1 do
11 |   |   |   ic ← col_ind[ip] ;
12 |   |   |   tag[color[ic]] ← seq;
13 |   |   end
14 |   end
15 |   flag_newcolor ← true ;
16 |   for jp ← 1 to maxgrp do
17 |   |   if tag[jp] ≠ seq then
18 |   |   |   flag_newcolor ← false;
19 |   |   end
20 |   end
21 |   if flag_newcolor = true then
22 |   |   maxgrp ← maxgrp + 1;
23 |   end
25 |   color[jcol] ← jp ;
26 end
```

Figure 3.4: Sequential Greedy Coloring

Figure 3.5 illustrates how *CRCS* data structure aids the *sequential greedy algorithm* to compute a coloring without explicitly constructing the intersection graph.

Besides *CRCS*, data structures needed for *sequential coloring* are : a temporary tagging array of size *n*, `tag` such that `tag[c] = j`, if and only if column *j* has color *c* assigned to it. The given ordering is stored in the input array of size *n*, named *order*. The given

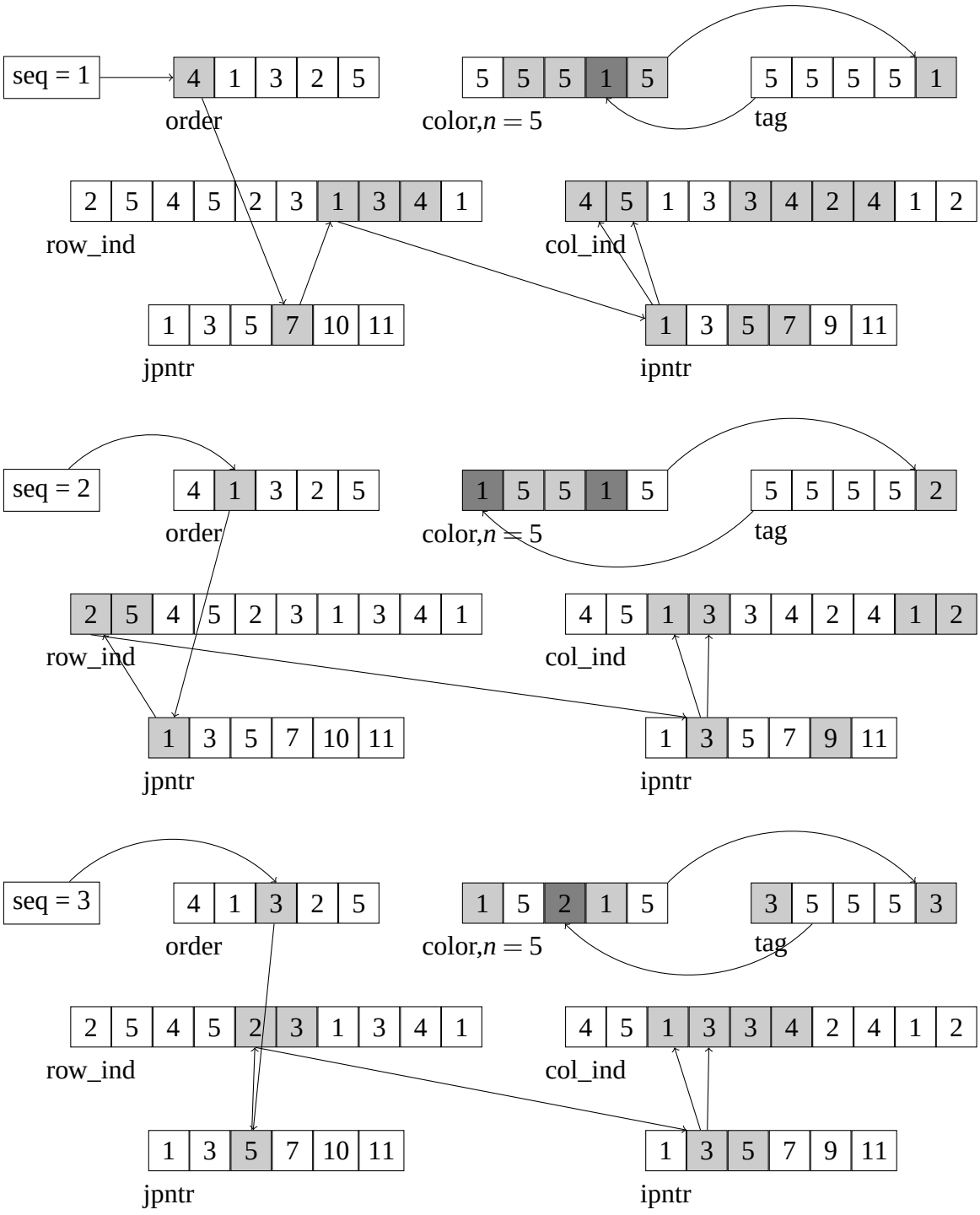

Figure 3.5: Sequential Coloring example

ordering for this example is $4,1,3,2,5$. *Sequential Coloring* puts the result in an integer array of size *n*, named *color*.

After initialization phase, in line 7, with $seq = 1$, we pick the column 4 for coloring. The related value of *jpntr* for this column is, 7 and 10. Using the indices $7,8,9$ in *jpntr* gives us the nonzero row entries for the current column, as *row_ind[7]*, *row_ind[8]*, and *row_ind[9]*, which happens to be $1,3$ and ,4.

As soon as we compute a row index for a nonzero entry in current column in line 9, we consult *ipntr* in the same way to find out the column indices for nonzero elements in the current row in lines $10-11$. As an example, first row indices found to be 1 with corresponding *ipntr[1]* and *ipntr[1+1]* values as 1 and 3, gives us column indices 4, and 5. These two columns form a clique in the intersection graph. In this way, we find the neighboring columns in *Neighborhood computation* phase.

In tagging phase, color value of column 5, *color[5]*, $n = 5$, is tagged by the current sequence value $seq = 1$. Similarly we find the other columns 3, and 2, both have the color value of 5, which is marked with current sequence value 1.

In coloring phase, scanning *tag* from the beginning will find first *tag* value which is not marked with current sequence value, which happens to be 1. We color the column 4 with color 1.

Once column 4 is colored, we increase *seq* value and pick the second column in the given *order* array (column 1). Similarly we consult *jpntr*, *row_ind*, *ipntr* and *col_ind* and compute the neighborhood as 2 and 3. Both have color 5, so we mark *tag[5] = 2*. Scanning *tag* from the beginning finds the first non-neighbor color in index 1, and we color it with color value of 1.

Similarly $seq = 3$ gives us column 3 with a neighborhood of 1 and 4, having colors 1. Scanning *tag* from the beginning gives us the first *non-seq* color as 2 and we assign this color to column 3.

In the same way we color column 2 with color value 2 and column 5 with color value 3.

3.1.1 Analysis

Lemma 1 *The greedy sequential coloring algorithm requires* $O(\sum_{i=1}^{m} \rho_i{}^2)$ *operations.*

Proof As stated previously in Figure 3.3, sequential algorithm can be broken into four major computational steps. Clearly, Initialization requires $O(n)$ steps. Number of operations needed for *Neighborhood computation* and *Tagging* is proportional to

$$\sum_{j=1}^{n} \sum_{\substack{a_{ij} \neq 0 \\ i=1}}^{m} \rho_i = \sum_{i=1}^{m} \rho_i{}^2. \tag{3.1}$$

We need not do more than $O(maxgrp) \leq O(deg(a_j)+1)$ steps to find the smallest unmarked color for column j. Therefore, the time complexity for *n* columns would be

$$O(\sum_{i=1}^{m} deg(j)+1) \leq O(\sum_{i=1}^{m} \rho_i{}^2). \tag{3.2}$$

3.2 Ordering Methods

3.2.1 Degree Calculation

Compute Degree method provides the necessary degree information for the ordering algorithms to function properly. Similar to *Sequential Greedy coloring*, *Compute Degree method* also uses only *CRCS* data structure. It calculates the degree information by visiting the adjacent columns for each column j, where $j \in \{1,\ldots,n\}$. Algorithm for degree calculation is given in Figure 3.6.

Algorithm 4: Compute Degree Algorithm

```
1  for j ← 1 to n do
2  |  ndeg[jp] ← 0 ;
3  |  tag[jp] ← 0 ;
4  end
5  for jcol ← 2 to n do
6  |  tag[jcol] ← n ;
7  |  for jp ← jpntr[jcol] to jpntr[jcol+1] − 1 do
8  |  |  ir ← row_ind[jp] ;
9  |  |  for ip ← ipntr[ir] to ipntr[ir+1] − 1 do
10 |  |  |  ic ← col_ind[ip] ;
11 |  |  |  if tag[ic] < jcol then
12 |  |  |  |  tag[ic] ← jcol ;
13 |  |  |  |  ndeg[ic] ← ndeg[ic] + 1 ;
14 |  |  |  |  ndeg[jcol] ← ndeg[jcol] + 1 ;
15 |  |  |  |  maxdeg ← max(ndeg[jcol], ndeg[ic], maxdeg) ;
16 |  |  |  end
17 |  |  end
18 |  end
19 end
```

Figure 3.6: Algorithm for Degree calculation

Traversing all the adjacent columns for each column j, results in $\sum_{\substack{a_{ij}\neq 0 \\ i=1}}^{m} \rho_i$ number of operations, where ρ_i is number of nonzero entries in a row i. Hence the complexity for ComputeDegree method is proportional to

$$\sum_{j=1}^{n} \sum_{\substack{a_{ij}\neq 0 \\ i=1}}^{m} \rho_i = \sum_{i=1}^{m} {\rho_i}^2. \tag{3.3}$$

3.2.2 Priority Queue

We often need to use a *priority queue* for our ordering algorithms described later in the chapter. In this section, we are going to describe the priority queue data structure and

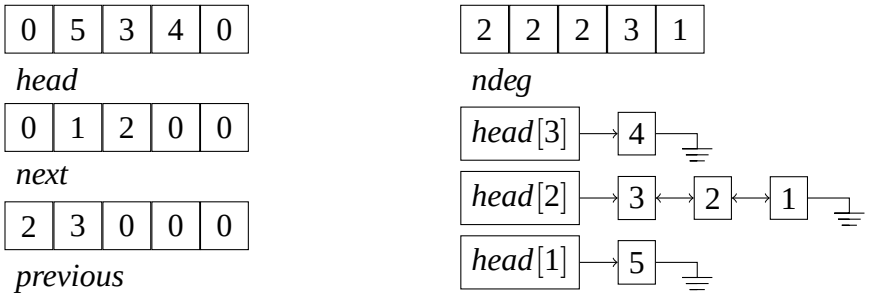

Figure 3.7: Bucket data structure example

related algorithms, as well as their asymptotic complexities.

Our priority queue is structured by *buckets* which is a common implementation for priority queues [14]. Given that the maximum priority is K, a priority queue can be implemented with K-array of pointers $head[]$, where $head[k]$ points to the kth bucket.

Each bucket is implemented as a *two-way linked list* for easy deletion and addition. Two additional integer arrays, *next* and *prev*, are used, where $next[j]$ is the element immediately following j in a bucket, and $prev[j]$ is the element immediately preceding j. If the next (or previous) element is empty, then $next[j] = 0$ (or $prev[j] = 0$). Figure 3.7 shows the data structures and a graphical representation of a priority queue. Note that functionality of *two-way linked list*s are accomplished by integer arrays only.

Build Priority Queue

Figure 3.8 shows the algorithm to build a priority queue given an integer array containing degree information (priority). Build Priority Queue Algorithm has a runtime complexity of $O(n)$.

Algorithm 5: Build Priority Queue

input : *ndeg*, an integer array of size *n*, containing degree information of columns $\{1 \ldots n\}$
input : *head*,*next*, and *prev*, integer arrays used for priority queue data structure
output: Priority Queue constructed in *head*, *next*, *previous*

```
1 for jp ← 1 to n do
2 |   numdeg ← ndeg[jp];
3 |   previous[jp] ← 0;
4 |   next[jp] ← head[numdeg];
5 |   if head[numdeg] > 0 then
6 |   |   previous[head[numdeg]] ← jp;
7 |   end
8 |   head[numdeg] ← jp;
9 end
```

Figure 3.8: Algorithm for initializing priority queue from degree information list

Add Column in Priority Queue

Algorithm for adding a column *jcol* with priority *numdeg* in a priority queue is given in Figure 3.9 and it has a complexity of $O(1)$.

Algorithm 6: Add a column in priority queue.

input : *head*,*next*, and *prev*, integer arrays used for priority queue data structure
output: Column *jcol* added in priority queue in appropriate location

```
1 previous[jcol] ← 0;
2 next[jcol] ← head[numdeg];
3 if head[numdeg] > 0 then
4 |   previous[head[numdeg]] ← jcol;
5 end
6 head[numdeg] ← jcol;
```

Figure 3.9: Algorithm for adding a column in a priority queue

Delete Column From Priority Queue

Algorithm for deleting a column *jcol* with priority *numdeg* from a priority queue has a complexity of $O(1)$ and is shown in Figure 3.10.

Algorithm 7: Delete a column from priority queue.

input : *head*,*next*, and *prev*, integer arrays used for priority queue data structure
output: Priority Queue with column *jcol* removed

```
1 if previous[jcol] = 0 then
2 |   head[numdeg] ← next[jcol];
3 else
4 |   next[previous[jcol]] ← next[jcol];
5 end
6 if next[jcol] > 0 then
7 |   previous[next[jcol]] ← previous[jcol];
8 end
```

Figure 3.10: Algorithm for deleting a column from a priority queue

3.2.3 Largest-First Ordering

Largest first ordering is the simplest of all the orderings. Sorting the vertices $V = \{v_1, v_2, \ldots, v_n\}$ in non-increasing degrees in G, represents largest first ordering.

Figure 3.11 describes the algorithm for Largest-First ordering. At the first phase the priority queue is constructed from the degree information computed by the *computeDegree* method, which take $O(n)$ number of steps. Then the priority queue is used to sort the vertices in *order* array in non-decreasing order of degree information. It also takes $O(n)$ number of steps. We can see that the running time of *Largest First Ordering* is dominated by the complexity of *computeDegree* method. So the runtime complexity of *LFO* is the same as *computeDegree*,

$$\sum_{j=1}^{n} \sum_{\substack{a_{ij}\neq 0 \\ i=1}}^{m} \rho_i = \sum_{i=1}^{m} {\rho_i}^2. \tag{3.4}$$

Algorithm 8: Largest First Ordering Algorithm

input : *ndeg*, an integer array of size *n*, containing degree information of columns $\{1 \ldots n\}$

output: *order*, an integer array of size *n*

```
1  maxdeg ← −1 ;
2  for jp ← 1 to n do
3  |   head[jp − 1] ← 0 ;
4  |   maxdeg ← max(maxdeg, ndeg[jp]) ;
5  end
6  buildPriorityQueue(n, ndeg, head, next, previous) ;
7  for numord ← 1 to n do
   |   /* choose a column jcol of maximal degree                    */
8  |   jcol ← 0;
9  |   while jcol ≤ 0 do
10 |   |   jcol ← head[maxdeg] ;
11 |   |   if jcol ≤ 0 then
12 |   |   |   maxdeg ← maxdeg − 1 ;
13 |   |   end
14 |   end
15 |   order[numord] ← jcol ;
16 |   if numord < n then
   |   |   /* Delete Jcol from the head of the list                 */
17 |   |   head[maxdeg] ← next[jcol] ;
18 |   |   if next[jcol] > 0 then
19 |   |   |   previous[next[jcol]] ← 0;
20 |   |   end
21 |   end
22 end
```

Figure 3.11: Largest First Ordering Algorithm

3.2.4 Smallest-Last Ordering

Assume the vertices $V' = \{v_n, v_{n-1}, \ldots, v_{i+1}\}$ have already been ordered. The i-th vertex in this order is an unordered vertex u such that $deg(u)$ is minimum in $G[V \setminus V']$.

Figure 3.12 shows the algorithm for Smallest-last ordering. The major computational steps for Smallest-last ordering algorithms are

1. **Initialization** Lines 1–6, $O(n)$ time is required for initializing of data structures including `tag` integer array, and constructing priority queue from degree information.

2. **Choosing a column** In lines 12-14, we remove a column j from the priority queue with the minimal degree in $G[V \setminus V']$, place it in the `order` array, and tag it, which requires $O(1)$ operation for a single column j.

3. **Neighborhood Computation** In lines, 16–19 we compute the neighborhood for a column j. As seen previously, searching neighborhood for a single column takes $O(\sum_{a_{ij} \neq 0} \rho_i)$ number of operations.

4. **Tagging and Updating of degree/priority** In lines 20–23 tagging and updating of degree is performed for the neighbors of column j, which takes $O(1)$ operations.

Choosing a Column and *Neighborhood Computation* executes for n times, which gives us a computational complexity of $O(n)$, and $\sum_{i=1}^{m} {\rho_i}^2$. As *Tagging and Computation* is done along with the same loop in neighborhood computation, they also execute for $\sum_{i=1}^{m} {\rho_i}^2$ number of times, resulting a computational complexity of

$$\sum_{i=1}^{m} {\rho_i}^2.$$

Algorithm 9: Smallest-last Ordering Algorithm

input : *ndeg*, an integer array of size *n*, containing a degree information for columns $\{1 \ldots n\}$

output: *order*, an integer array of size *n*

```
1  mindeg ← n;
2  BuildPriorityQueue(ndeg, head, next, previous);
3  for jp ← 1 to n do
4  |   tag[jp] ← n;
5  |   mindeg ← min(mindeg, ndeg[jp]);
6  end
7  maximalClique ← 0;
8  for numord ← n to 1 do
9  |   if (mindeg + 1 = numord) and (maximalClique = 0) then
10 |   |   maximalClique ← numord;
11 |   end
   |   /* find column jcol with minimal degree                    */
12 |   (jcol, mindeg) ← ExtractMin();
13 |   order[numord] ← jcol;
14 |   tag[jcol] ← 0;
15 |   if numord > 1 then
16 |   |   for jp ← jpntr[jcol] to jpntr[jcol + 1] − 1 do
17 |   |   |   ir ← row_ind[jp];
18 |   |   |   for ip ← ipntr[ir] to ipntr[ir + 1] − 1 do
19 |   |   |   |   ic ← col_ind[ip];
20 |   |   |   |   if tag[ic] > numord then
21 |   |   |   |   |   tag[ic] ← numord;
22 |   |   |   |   |   numdeg ← DecreaseDegree(ic);
23 |   |   |   |   |   mindeg ← min(mindeg, numdeg);
24 |   |   |   |   end
25 |   |   |   end
26 |   |   end
27 |   end
28 end
```

Figure 3.12: Smallest-Last Ordering Algorithm

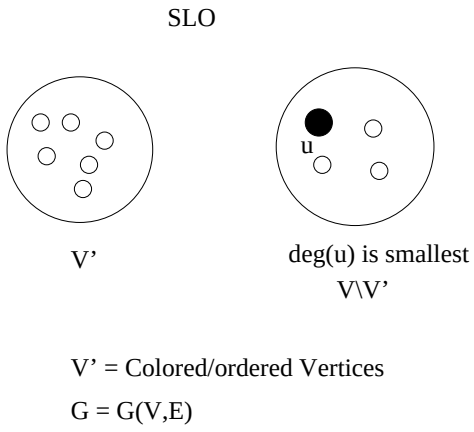

Figure 3.13: Smallest-Last Greedy Coloring

3.2.5 Incidence-Degree Ordering

Assume the vertices $V' = \{v_1, v_2, \ldots, v_{i-1}\}$ have been ordered. The i-th vertex in this order is an unordered vertex u such that $deg(u)$ is maximum in $G[V']$. Ties are broken by choosing the vertex that has largest degree in G . The algorithm is shown in Figure 3.16

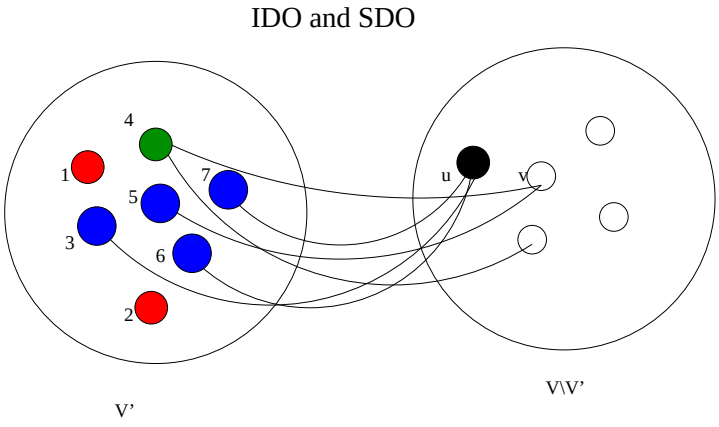

Figure 3.14: SD and ID Greedy Coloring

Incidence Degree Ordering is similar to *Smallest Largest First Ordering*, and the largest

computational cost is incurred by search for adjacent columns of chosen column j in each step. Hence the complexity of *ido* is

$$\sum_{i=1}^{m} \rho_i^2$$

Index Sort

Index sort is used in Incidence Degree ordering, and the complexity for this algorithm is $O(n)$. The algorithm is given in Figure 3.15.

Algorithm 10: Index Sort Algorithm

input : *ndeg*, an integer array of size *n*, containing degree information of columns $\{1 \ldots n\}$.
mode indicates the desired sorting
output: Sorted *index*, with *last*, *next* according to *mode*

```
1  for i ← 0 to nmax do
2  |  last[i] ← 0;
3  end
4  for k ← 1 to n do
5  |  l ← num[k];
6  |  next[k] ← last[l];
7  |  last[l] ← k;
8  end
9  if mode = 0 then
10 |  return;
11 end
   /* store the pointers to the sorted array in index.          */
12 i ← 1;
13 if mode > 0 then
14 |  jl ← 0;
15 |  ju ← nmax;
16 else
17 |  jl ← nmax;
18 |  ju ← 0;
19 end
20 for j ← jl to ju do
21 |  k ← last[j];
22 |  while k ≠ 0 do
23 |  |  index[i] ← k;
24 |  |  i ← i + 1;
25 |  |  k ← next[k];
26 |  end
27 end
```

Figure 3.15: Index Sort Algorithm

Algorithm 11: Incidence-Degree Ordering Algorithm

input : *ndeg*, an integer array of size *n*, containing a degree information for columns $\{1 \ldots n\}$

output: *order*, an integer array of size *n*

```
1  indexsort(n, n−1, ndeg, −1, tag, previous, next);
2  BuildPriorityQueue(head, next, previous, tag);
3  Initialization();
4  ComputeMaximumSearchLength();
5  for numord ← 1 to n do
6  |  UpdateMaximalClique();
   |  /* choose a column jcol of maximal incidence degree          */
7  |  j ← ChooseColumnJWithMaximumSatDeg();
8  |  j ← SearchMaxLenghtToFindBetterChoice();
9  |  order[j] ← numord;
10 |  numord ← numord + 1;
11 |  if numord ≤ N then
12 |  |  deleteColumn(head, next, previous, maxinc, j);
13 |  |  tag[j] ← n;
14 |  |  forall the j' ∈ adj(j) do
15 |  |  |  if tag[j] < numord then
16 |  |  |  |  tag[j] ← numord;
17 |  |  |  |  incidence ← order[j'];
18 |  |  |  |  order[j'] ← order[j'] + 1;
19 |  |  |  |  deleteColumn(head, next, previous, incidence, j');
20 |  |  |  |  addColumn(head, next, previous, incidence + 1, j');
21 |  |  |  end
22 |  |  end
23 |  end
24 end
25 for jcol ← 1 to n do
26 |  previous[order[jcol]] ← jcol;
27 end
28 for jp ← 1 to n do
29 |  order[jp] ← previous[jp];
30 end
```

Figure 3.16: Incidence-Degree Ordering Algorithm

3.2.6 Saturation-Degree Ordering

Assume the vertices $V' = \{v_1, v_2, \ldots, v_{i-1}\}$ have been ordered and colored. The *i*-th vertex in *Saturation-Degree ordering* is an unordered vertex *u* such that $kdeg(u)$ is largest in $G[V \setminus V']$. Ties are broken by choosing the vertex that has the largest degree in $G(V \setminus V')$.

Figure 3.17 describes the algorithm for saturation degree ordering algorithm. The major computational steps in the algorithm are :

1. **Initialization** Lines 1-2, $O(n)$ time is required for initialization of necessary data structures and construct the priority queue.

2. **Choosing a Column** Line 4, Choosing the next column to color is done in $O(\delta_j \delta_{max})$ time, where δ_{max} is the maximum degree in $G(V)$, and δ_j is the degree of column j in $G(V \setminus V')$.

3. **Finding Smallest Unmarked Color** Line 4. We search the neighborhood of chosen column j, and mark all the corresponding colors in $O(\sum_{\substack{i \\ a_{ij} \neq 0}} \rho_i)$ steps. To find the smallest color it does not take more than $O(\delta_j)$ operations.

4. **Neighborhood Computation** Lines 9-19, updates the saturation degree and induced degree of the adjacent vertices of column j. It takes $O(\sum_{\substack{i \\ a_{ij} \neq 0}} \rho_i)$ time to search the neighborhood of column j.

5. **Tagging and Updating of Degree/Priority** In Lines 10-18 tagging and updating of degree for each adjacent column is done. Tagging is done in line 11. To find out whether the color chosen for column j exists in the neighborhood of j, we consult a simple *bitset* table with the dimension of $color \times n$ in line 12. If the color assigned to column j introduces a new color in the neighborhood of column $j' \in adj(j)$, we increase the saturation degree of column j' in line 14.

Complexity of *Saturation-degree* ordering has been found to be $O(\delta_{max} \sum_{i=1}^{m} \rho_i{}^2)$ as computed as follows:

$$\begin{aligned}
\sum_{j=1}^{n} \delta_{max}\delta j + \sum_{j=1}^{n} \sum_{\substack{i=1 \\ a_{ij}\neq 0}}^{m} \rho_{i'} &= \sum_{j=1}^{n} \delta_{max}\delta j + \sum_{i=1}^{m} \rho_i{}^2 \\
&= \delta_{max} \sum_{j=1}^{n} \delta j + \sum_{i=1}^{m} \rho_i{}^2 \\
&\leq \delta_{max} \sum_{i=1}^{m} \rho_i{}^2 + \sum_{i=1}^{m} \rho_i{}^2 \\
&= \delta_{max} \sum_{i=1}^{m} \rho_i{}^2 \qquad (3.5)
\end{aligned}$$

Algorithm 12: Saturation Degree Ordering Algorithm

input : *ndeg*, an integer array of size *n*, containing a degree information for columns $\{1 \ldots n\}$

output: *color*, an integer array of size *n*

```
1  BuildPriorityQueue(head, next, previous, ndeg, tag);
2  Initialization();
3  for numord ← 1 to n do
4  |   Find a column j with maximum saturation degree and induced degree;
5  |   Find smallest feasible color col for column j;
6  |   color[j] ← col;
7  |   numord ← numord + 1;
8  |   tag[j] ← n;
9  |   forall the j' ∈ adj(j) do
10 |   |   if tag[j'] < numord then
11 |   |   |   tag[j'] ← numord;
12 |   |   |   if bitset[col][j'] = false then
13 |   |   |   |   bitset[col][j'] ← true;
14 |   |   |   |   satDeg[j'] ← satDeg[j'] + 1;
15 |   |   |   end
16 |   |   |   inducedDeg[j'] ← inducedDeg[j'] − 1;
17 |   |   |   update priority of column j' in the priority queue;
18 |   |   end
19 |   end
20 end
```

Figure 3.17: Saturation Degree Ordering Algorithm

3.2.7 Recursive-Largest-First Coloring

Recursive-largest-first(RLF) algorithm partitions the vertex set V into $V_1, V_2, \ldots, V_p$ independent sets, and constructs a structurally orthogonal column partition with p number of column groups.

The first vertex of V_i is chosen in a way such that it has the largest degree in $V \setminus \bigcup_{j=1}^{i-1} V_j$ induced graph, adjacent vertices of the chosen vertex are added to the inadmissible set U. RLF continues adding vertices to the independent set V_i, by choosing v_k which has the largest number of adjacent vertices in the set U at k-th step, and neighbors of v_k are also added to U. Figure 3.18 gives an overview of RLF, and Figure 3.19 describes the algorithm.

Algorithm 1 Recursive Largest First Algorithm

1. Initialize $U = \phi$, $C = \phi$, $V' = V$, and $q = 0$.
2. Choose the vertex v_k with the maximum degree in V'. Increment q and proceed to 3.
3. Color v_k with color q and move it from V' to C. Find all the adjacent vertices of v_k and move them from V' to U. If V' is not empty then proceed to 4. If $C = V$ then exit. Otherwise, move the vertices from U to V', and proceed to 2.
4. Choose a vertex from V' which has the maximum number of adjacent vertices in U. Goto 3

Figure 3.18: Overview of Recursive Largest First Algorithm

We identify the major computational steps for Recursive Largest First algorithm from Figure 3.19 as:

1. **Initialization** Lines 1-3 performs initialization and it takes $O(n)$ operations.
2. **Choosing a Column** Lines 5-10 chooses a column to be inserted into the current color class. It takes $O(maxdeg)$ operations.

3. **Coloring** Line 11, Coloring and tagging takes $O(1)$ operation, as we have a predefined color already.

4. **Neighborhood Computation** Lines 16-31 computes the distance 2 neighborhood for column j, moves columns from V' to U, updates the degree information in the priority queue.

5. **Reinitialization** Lines 32-37. If the set of admissible columns V' is empty, we reinitialize the data structures needed, and start constructing a new color class. It requires $O(n)$ operations.

Time Complexity of RLF ordering can be computed to be $O(p\kappa_{max}\sum_{i=1}^{m}\rho_i{}^2)$ as follows, where κ_{max} is the maximum number of nonzeroes in a column, and p is the number of partitions:

$$\begin{aligned}
\sum_{q=1}^{p}\sum_{i=1}^{m}\sum_{\substack{j=1\\a_{ij}\neq 0}}^{n}\sum_{\substack{i'=1\\a_{i'j}\neq 0}}^{m}\rho_{i'} &= \delta_{max}\sum_{i=1}^{m}\rho_i\sum_{j=1}^{n}\sum_{i'=1}^{m}[a_{ij}\neq 0][a_{i'j}\neq 0]\\
&= p\sum_{i=1}^{m}\rho_i\sum_{j=1}^{n}[a_{ij}\neq 0]\sum_{i'=1}^{m}[a_{i'j}\neq 0]\\
&= p\sum_{i=1}^{m}\rho_i\sum_{j=1}^{n}[a_{ij}\neq 0]\kappa_j\\
&\leq p\sum_{i=1}^{m}\rho_i\sum_{j=1}^{n}[a_{ij}\neq 0]\kappa_{max}\\
&= p\kappa_{max}\sum_{i=1}^{m}\rho_i{}^2 \qquad (3.6)
\end{aligned}$$

Algorithm 13: Recursive Largest First Algorithm

```
1  Initialization();
2  BuildPriorityQueue(head, next, previous, ndeg);
3  BuildPriorityQueue(uhead, unext, uprevious, 0);
4  for numord ← 1 to n do
5      if newColorClass = true then
6          newColorClass = false;
7          (j, maxdeg) = FindMaxFromPriorityQueue(V);
8      else
9          (j, u_maxdeg) = FindMaxFromPriorityQueue(U);
10     end
11     color[j] ← q;    tag[j] ← n;
12     if numord = n then
13         break;
14     end
15     DeleteColumn(j, V);    DeleteColumn(j, U);
16     forall the j' ∈ adj(j) do
17         if tag[j'] < numord then
18             tag[j'] ← numord;
19             priority_queue.decrease(j', V);
20             if j' ∉ U then
21                 U ← U + {j'};
22                 u_queue.remove(j');
23                 forall the j'' ∈ adj(j') do
24                     if j'' ∈ V and u_tag[j''] ≠ j' then
25                         u_tag[j''] ← j';
26                         u_queue.increase(j'');
27                     end
28                 end
29             end
30         end
31     end
32     if V = φ then
33         q ← q + 1;
34         newColorClass = true;
35         Reinitialize V, U and u_tag;
36         BuildPriorityQueue(U);
37     end
38 end
```

Figure 3.19: Recursive Largest First Algorithm

3.2.8 RLF-SLO coloring

RLF-SLO is a parametrized hybrid coloring based on *RLF* and *SLO*. This algorithm runs *RLF* coloring on first p columns, then switches to *SLO* ordering and colors the remaining $(n-p)$ columns. The motivation behind this hybrid approach is to get the better coloring results of *RLF* with better timing results of *SLO*.

3.3 Storage Format

We provide necessary code with examples to read the matrix descriptions from Matrix Market Exchange Format files. As DSJM depends on the client code to supply the matrix description, user can use any other suitable file formats such as *Harwell-Boeing Exchange Format*.

Chapter 4

Computational Experiments

In this chapter we present computational results for the algorithms implemented in DSJM toolkit. In Section 4.2 we present the data sets for our experiments. Numerical results are given in Section 4.3. Tables listing the numbers of colors obtained can be found in Section 4.3.1. We present the experimental results for hybrid coloring in Section 4.4

4.1 Test Environment

Experiments were done on an IBM PC with 2.8 GHz Intel Pentium CPU, 1 GB RAM, and 512 KB L2 cache running 32-bit Linux.

4.2 Data Sets

We describe two different sets of sparse matrices for our experimental results. Table 4.1 lists the first set of matrices along with their structural properties. The test matrices were obtained from the University of Florida Sparse Matrix Collection [10]. These matrices were also reported for presenting experimental results in [16]. Matrix af23560 is a computational fluid dynamics problem. Matrices with the prefix *lp* in their names are linear programming problems from Netlib. The *lhr*-matrices come from chemical process simulation problems. The *cage*-matrices are models used in DNA electrophoresis. Matrices e30r2000 and e40r0100 arise in modeling 2D fluid flow.

Columns labeled *m*, *n*, and *nnz* denote the number of rows, columns and non-zeroes respectively, in the matrices. ρ_{max} and ρ_{min} are maximum and minimum number of non-zeroes in any row, and $\bar{\rho}$ is arithmetic mean of number of non-zeroes in each row.

Table 4.2 lists matrices from Harwell-Boeing test matrices [25, 1, 2] and the University

Table 4.1: Matrix Statistics for Set 1

Matrix Name	m	n	nnz	ρ_{max}	$\bar{\rho}$	ρ_{min}
af23560	23560	23560	484256	21	20	11
cage11	39082	39082	559722	31	14	3
cage12	130228	130228	2032536	33	15	5
e30r2000	9661	9661	306356	62	31	8
e40r0100	17281	17281	553956	62	32	8
lhr10	10672	10672	232633	63	21	1
lhr14	14270	14270	307858	63	21	1
lhr34	35152	35152	764014	63	21	1
lhr71c	70304	70304	1528092	63	21	1
lpcrea	3516	7248	18168	360	5	1
lpcreb	9648	77137	260785	844	27	1
lpcred	8926	73948	246614	808	27	1
lpfit2d	25	10524	129042	10500	5161	1427
lpdfl001	6071	12230	35632	228	5	2
lpken11	14694	21349	49058	122	3	1
lpken13	28632	42659	97246	170	3	1
lpken18	105127	154699	358171	325	3	1
lpmarosr7	3136	9408	144848	48	46	6
lppds10	16558	49932	107605	96	6	18
lppds20	33874	108175	232647	96	6	1
lpstocfor3	16675	23541	76473	15	4	1

of Florida Matrix Collection [10] translated from Netlib [3]. These matrices were also reported in [18] to present computational results.

Table 4.2: Matrix Statistics for Set 2

Matrix Name	m	n	nnz	ρ_{max}	$\bar{\rho}$	ρ_{min}
abb313	313	176	1557	6	4	1
adlittle	56	138	424	27	7	1
agg	488	615	2862	19	5	2
agg2	516	758	4740	49	9	2
agg3	516	758	4756	49	9	2
arc130	130	130	1282	124	9	1
ash219	219	85	438	2	2	2
ash292	292	292	2208	14	7	4
ash331	331	104	662	2	2	2
ash608	608	188	1216	2	2	2
ash958	958	292	1916	2	2	2
blend	74	114	522	29	7	2
bore3d	233	334	1448	73	6	1
bp0	822	822	3276	266	3	1
bp1000	822	822	4661	308	5	1
bp1200	822	822	4726	311	5	1
bp1400	822	822	4790	311	5	1
bp1600	822	822	4841	304	5	1
bp200	822	822	3802	283	4	1
bp400	822	822	4028	295	4	1
bp600	822	822	4172	302	5	1
bp800	822	822	4534	304	5	1
can1054	1054	1054	12196	35	11	6
can1072	1072	1072	12444	35	11	6
can256	256	256	2916	83	11	4
can268	268	268	3082	37	11	4
can292	292	292	2540	35	8	4
can634	634	634	7228	28	11	2
can715	715	715	6665	105	9	2
curtis54	54	54	291	12	5	3
dwt1007	1007	1007	8575	10	8	3
dwt1242	1242	1242	10426	12	8	2
dwt2680	2680	2680	25026	19	9	4
dwt419	419	419	3563	13	8	6
dwt59	59	59	267	6	4	2
eris1176	1176	1176	18552	99	15	2
fs5411	541	541	4285	11	7	1
fs5412	541	541	4285	11	7	1

Continued on next page . . .

Table 4.2: Matrix Statistics for Set 2 (Continued)

Matrix Name	m	n	nnz	ρ_{max}	$\bar{\rho}$	ρ_{min}
gent113	113	113	655	20	5	1
ibm32	32	32	126	8	3	2
impcola	207	207	572	8	2	1
impcolb	59	59	312	7	5	2
impcolc	137	137	411	8	3	1
impcold	425	425	1339	10	3	1
impcole	225	225	1308	12	5	1
israel	174	316	2443	119	14	2
lunda	147	147	2449	21	16	5
lundb	147	147	2441	21	16	5
scagr25	471	671	1725	10	3	1
scagr7	129	185	465	10	3	1
shl0	663	663	1687	422	2	1
shl200	663	663	1726	440	2	1
shl400	663	663	1712	426	2	1
stair	356	614	4003	36	11	2
standata	359	1274	3230	745	8	2
str0	363	363	2454	34	6	1
str200	363	363	3068	30	8	1
str400	363	363	3157	33	8	1
tuff	333	628	4561	113	13	0
vtpbase	198	346	1051	38	5	1
watt2	1856	1856	11550	128	6	1
west0067	67	67	294	6	4	1
west0381	381	381	2157	25	5	1
west0497	497	497	1727	28	3	1
will199	199	199	701	6	3	1
will57	57	57	281	11	4	2

4.3 Numerical Results

4.3.1 Partitioning Results

Table 4.3 lists the number of structurally orthogonal groups achieved by DSJM for each constructive heuristics for data set 1. On the left side of the table we list the name of the

matrices and their structural properties. On the right side, we list the number of colors obtained for each constructive heuristics in their respective columns. Table 4.4 lists the number of colors obtained for data set 2 respectively The number in boldface represents the best(smallest) partitioning(coloring) for the respective problem instance.

ρ_{max} is a lower bound of the number of groups in a structurally orthogonal partition of the columns. Though a maximal clique in a graph can be a weak lower bound, we found ρ_{max} to be a good one. The ordering algorithms SLO and IDO find a maximal clique as a by-product in the column intersection graph $G(A)$. We list maximal clique larger than ρ_{max}, in parentheses in the ρ_{max} column. We observed that we often cannot find clique which is larger in size than ρ_{max}, and in many cases ρ_{max} proves to be optimal number of groups in a structurally orthogonal partition of the columns. We found ρ_{max} to be optimal for 15 matrices out of 21 in data set 1. For data set 2, it was true for 49 matrices out of 65. Moreover, the summation of exact coloring values for the matrices in data set 2 was computed in [18], and found to be 6447.The summation of ρ_{max} is 6408 over all the matrices. So, we consider ρ_{max} as a good lower bound for the test data sets, as well as in practice.

RLF produced the best partitioning in 19 out of 21 problem instances for data set 1, with optimal coloring for 14 for them. We have observed that RLF produces better coloring when there is a larger gap between number of colors and known lower bound. For example, for matrices *af23560*, *cage11*, *cage12* and *lpmarosr7* the number of colors obtained is lower than the known lower bound by 35. For test data set 1, RLF produced 2.05 fewer colors on average compared to other heuristics. For the above mentioned four matrices, RLF produced 9 fewer colors on average. Total number of smallest structurally orthogonal column groups over the test instances for the ordering algorithms are 14170, while RLF produced 14172 colors.

On test set 2, RLF is as good as any of the other ordering and outperformed the other

Table 4.3: Coloring Results using DSJM for Data Set 1

Matrix Name	*m*	*n*	*nnz*	ρ_{max}	*RLF*	*IDO*	*SLO*	*LFO*	*SDO*
af23560	23560	23560	484256	21 (30)	**37**	43	41	44	41
cage11	39082	39082	559722	31	**54**	65	62	68	59
cage12	130228	130228	2032536	33	**56**	70	68	72	60
e30r2000	9661	9661	306356	62	**65**	72	70	66	70
e40r0100	17281	17281	553956	62	67	70	71	**66**	68
lhr10	10672	10672	232633	63	64	64	**63**	64	**63**
lhr14	14270	14270	307858	63	**63**	64	**63**	64	**63**
lhr34	35152	35152	764014	63	**63**	64	**63**	64	**63**
lhr71c	70304	70304	1528092	63	**63**	64	**63**	64	**63**
lpcrea	3516	7248	18168	360	**360**	**360**	**360**	**360**	**360**
lpcreb	9648	77137	260785	844	**844**	**844**	845	**844**	**844**
lpcred	8926	73948	246614	808	**808**	**808**	**808**	**808**	**808**
lpfit2d	25	10524	129042	10500	**10500**	**10500**	**10500**	**10500**	**10500**
lpdfl001	6071	12230	35632	228	**228**	**228**	**228**	**228**	**228**
lpken11	14694	21349	49058	122	**122**	123	125	128	**122**
lpken13	28632	42659	97246	170	**170**	**170**	171	174	**170**
lpken18	105127	154699	358171	325	**325**	326	**325**	328	**325**
lpmarosr7	3136	9408	144848	48 (62)	**76**	85	83	100	90
lppds10	16558	49932	107605	96	**96**	**96**	**96**	**96**	**96**
lppds20	33874	108175	232647	96	**96**	**96**	**96**	**96**	**96**
lpstocfor3	16675	23541	76473	15	**15**	**15**	**15**	**15**	**15**
Total				14073	14172	14227	14216	14249	14204

ordering on 11 of the instances. Total number of smallest structurally orthogonal column groups over the test instances for the ordering algorithms are 6453.

Table 4.4: Coloring Results using DSJM for Data Set 2

Matrix Name	*m*	*n*	*nnz*	ρ_{max}	*RLF*	*IDO*	*SLO*	*LFO*	*SDO*
abb313	313	176	1557	6 (10)	**10**	11	**10**	11	11
adlittle	56	138	424	27	**27**	**27**	**27**	**27**	**27**
agg	488	615	2862	19	**19**	**19**	20	21	**19**
agg2	516	758	4740	49	**49**	50	**49**	50	**49**
agg3	516	758	4756	49	**49**	50	**49**	50	**49**
arc130	130	130	1282	124	**124**	**124**	**124**	**124**	**124**
ash219	219	85	438	2 (4)	**4**	**4**	**4**	5	**4**
ash292	292	292	2208	14	**14**	**14**	**14**	16	**14**
ash331	331	104	662	2 (6)	**6**	**6**	**6**	**6**	**6**
ash608	608	188	1216	2 (5)	**6**	**6**	**6**	**6**	**6**
ash958	958	292	1916	2 (6)	**6**	**6**	**6**	**6**	**6**
blend	74	114	522	29	**29**	**29**	**29**	**29**	**29**
bore3d	233	334	1448	73	**73**	**73**	**73**	**73**	**73**
bp0	822	822	3276	266	**266**	**266**	**266**	**266**	**266**
bp1000	822	822	4661	308	**308**	**308**	**308**	**308**	**308**
bp1200	822	822	4726	311	**311**	**311**	**311**	**311**	**311**
bp1400	822	822	4790	311	**311**	**311**	**311**	**311**	**311**
bp1600	822	822	4841	304	**304**	**304**	**304**	**304**	**304**
bp200	822	822	3802	283	**283**	**283**	**283**	**283**	**283**
bp400	822	822	4028	295	**295**	**295**	**295**	**295**	**295**
bp600	822	822	4172	302	**302**	**302**	**302**	**302**	**302**
bp800	822	822	4534	304	**304**	**304**	**304**	**304**	**304**
can1054	1054	1054	12196	35	**35**	**35**	**35**	**35**	**35**
can1072	1072	1072	12444	35	**35**	**35**	**35**	**35**	**35**
can256	256	256	2916	83	**83**	**83**	**83**	**83**	**83**
can268	268	268	3082	37	**37**	**37**	**37**	**37**	**37**
can292	292	292	2540	35	**35**	**35**	**35**	**35**	**35**
can634	634	634	7228	28	**28**	**28**	**28**	30	**28**
can715	715	715	6665	105	**105**	**105**	**105**	**105**	**105**
curtis54	54	54	291	12	**12**	**12**	**12**	**12**	**12**
dwt1007	1007	1007	8575	10	**10**	12	11	12	**10**
dwt1242	1242	1242	10426	12	**13**	14	14	15	**13**
dwt2680	2680	2680	25026	19	**19**	**19**	**19**	21	**19**
dwt419	419	419	3563	13 (14)	**15**	16	16	17	**15**
dwt59	59	59	267	6	**6**	**6**	7	7	**6**
eris1176	1176	1176	18552	99	**99**	**99**	**99**	**99**	**99**
fs5411	541	541	4285	11	**12**	13	13	14	**12**
fs5412	541	541	4285	11	**12**	13	13	14	**12**

Continued on next page

Table 4.4: Coloring Results using DSJM for Data Set 2 (Continued)

Matrix Name	*m*	*n*	*nnz*	ρ_{max}	*RLF*	*IDO*	*SLO*	*LFO*	*SDO*
gent113	113	113	655	20	**20**	**20**	**20**	**20**	**20**
ibm32	32	32	126	8	**8**	**8**	**8**	**8**	**8**
impcola	207	207	572	8	**8**	**8**	**8**	**8**	**8**
impcolb	59	59	312	7 (10)	**10**	11	11	11	**10**
impcolc	137	137	411	8	**8**	**8**	**8**	9	**8**
impcold	425	425	1339	10	**10**	11	11	12	**10**
impcole	225	225	1308	12 (20)	**21**	**21**	**21**	**21**	**21**
israel	174	316	2443	119	**119**	**119**	**119**	**119**	**119**
lunda	147	147	2449	21	22	24	24	27	**21**
lundb	147	147	2441	21	23	24	24	27	**22**
scagr25	471	671	1725	10	**10**	**10**	**10**	**10**	**10**
scagr7	129	185	465	10	**10**	**10**	**10**	**10**	**10**
shl0	663	663	1687	422	**422**	**422**	**422**	**422**	**422**
shl200	663	663	1726	440	**440**	**440**	**440**	**440**	**440**
shl400	663	663	1712	426	**426**	**426**	**426**	**426**	**426**
stair	356	614	4003	36	**36**	**36**	**36**	**36**	**36**
standata	359	1274	3230	745	**745**	**745**	**745**	**745**	**745**
str0	363	363	2454	34	**34**	**34**	**34**	**34**	**34**
str200	363	363	3068	30	**30**	**30**	**30**	**30**	**30**
str400	363	363	3157	33	**33**	**33**	**33**	**33**	**33**
tuff	333	628	4561	113	**114**	**114**	**114**	**114**	**114**
vtpbase	198	346	1051	38	**38**	**38**	**38**	**38**	**38**
watt2	1856	1856	11550	128	**128**	**128**	**128**	**128**	**128**
west0067	67	67	294	6 (7)	**8**	9	9	9	**8**
west0381	381	381	2157	25 (27)	**28**	29	30	29	**28**
west0497	497	497	1727	28	**28**	**28**	**28**	**28**	**28**
will199	199	199	701	6 (7)	**7**	**7**	**7**	8	**7**
will57	57	57	281	11	**11**	**11**	**11**	**11**	**11**
Total				6408	6453	6459	6468	6492	6452

4.3.2 Significance of fewer function evaluations in Jacobian Matrix Computation

Most of the time, Jacobian is computed as part of another iterative method. We have stated earlier in Chapter 1, how Jacobian computation is a part of Newton's method. Since Jacobian is computed in each iteration, the number of saved function evaluation from fewer

color groups can add up to a significant performance gain in the context of the iterative method. We present results of Newton's method from experiments run by Bouaricha and Schnabel [4] in Table 4.5. The Newton's algorithm used in their experiment computed Jacobian once in each iteration, and uses finite differencing method to do so. In the first and second column of the table, we list the number of iterations and function evaluations needed to solve the problems. Detail of the problems and the algorithm used can be found in [4]. In the third and fourth column, we calculate a hypothetical improvement if we could have achieved one and two less function evaluations, respectively, in each iteration. From the table, we can see that, even one less function evaluations can lead up to 24% performance gain.

4.3.3 Running Time

Table 4.6 lists the running time of each constructive heuristics implemented in DSJM. The experiments were run on a dedicated machine with minimal system load. Moreover the running time reported is the average of 5 runs of the respective ordering algorithm, to reduce any variation incurred by a sudden spike of increased usage of the CPU. We have tried to follow the instructions from [12] to gain the best performance from the computer system. It includes CPU time for both ordering and sequential algorithm. Reported time discards the running time for I/O operations (e.g reading the matrix description from file). The left side of the table contains the name of the matrices and the structural properties. On the right side, we list running time in seconds for each ordering algorithm in their respected columns. The smaller size of matrices in data set 2 results in very short running time, so we refrain us to report the running time for data set 2.

Table 4.5: Experimental results for Newton's Method

Matrix Dimension n	Iterations i	Fevals F	Improvement $\frac{i*l}{F}$ $l=1$	$l=2$
LTS problem	24	467	5.14%	10.28%
313	46	866	5.31%	10.62%
GRST problem	50	831	6.02%	12.03%
324	68	1065	6.38%	12.77%
	72	1176	6.12%	12.24%
LGNDR problem	74	375	19.73%	39.47%
50	75	381	19.69%	39.37%
Trigonometric problem	28	425	6.59%	13.18%
300	18	225	8.00%	16.00%
	66	939	7.03%	14.06%
Broyden banded problem	22	184	11.96%	23.91%
300	37	321	11.53%	23.05%
	44	411	10.71%	21.41%
Broyden tridiagonal problem	14	60	23.33%	46.67%
300	27	112	24.11%	48.21%
	31	151	20.53%	41.06%
Variable dimension problem	24	7525	0.32%	0.64%
300	44	13546	0.32%	0.65%
	44	13546	0.32%	0.65%
Distillation column problem	5	72	6.94%	13.89%
31	19	280	6.79%	13.57%
	26	357	7.28%	14.57%
Distillation column problem	8	136	5.88%	11.76%
99	20	315	6.35%	12.70%
	26	436	5.96%	11.93%

This tables presents the number of Iterations and Function evaluations for Newton's Method for some known problems from Bouaricha and Schnabel's experiments [4] and calculates a hypothetical improvements if fewer function evaluations would have been achieved in each iteration.

Table 4.6: Timing Results using DSJM for Data Set 1

Matrix Name	m	n	nnz	ρ_{max}	RLF	IDO	SLO	LFO	SDO
af23560	23560	23560	484256	21	4.84	0.54	0.52	0.36	0.88
cage11	39082	39082	559722	31	8.58	0.69	0.69	0.43	1.33
cage12	130228	130228	2032536	33	54.35	3.97	3.92	2.07	6.34
e30r2000	9661	9661	306356	62	5.09	0.50	0.50	0.37	0.72
e40r0100	17281	17281	553956	62	9.30	0.92	0.91	0.68	1.30
lhr10	10672	10672	232633	63	1.26	0.48	0.47	0.37	0.62
lhr14	14270	14270	307858	63	1.68	0.63	0.63	0.48	0.82
lhr34	35152	35152	764014	63	4.14	1.58	1.57	1.20	2.04
lhr71c	70304	70304	1528092	63	8.30	3.15	3.12	2.40	4.08
lpcrea	3516	7248	18168	360	1.22	0.05	0.04	0.03	0.18
lpcreb	9648	77137	260785	844	244.83	3.51	3.41	1.83	12.14
lpcred	8926	73948	246614	808	251.88	3.58	3.47	1.84	12.22
lpdfl001	6071	12230	35632	228	0.67	0.07	0.06	0.03	0.22
lpken11	14694	21349	49058	122	0.63	0.10	0.09	0.05	0.29
lpken13	28632	42659	97246	170	1.92	0.25	0.24	0.13	0.75
lpken18	105127	154699	358171	325	17.91	1.97	1.82	0.92	5.27
lpmarosr7	3136	9408	144848	48	4.26	0.32	0.30	0.21	0.51
lppds10	16558	49932	107605	96	0.87	0.15	0.14	0.08	0.43
lppds20	33874	108175	232647	96	2.04	0.34	0.34	0.19	0.97
lpstocfor3	16675	23541	76473	15	0.17	0.05	0.05	0.03	0.10

Table 4.7: Timing Results for Incidence Degree Partitioning.

Matrix Name	*m*	*n*	*nnz*	ColPack		DSJM	
				ot	pt	ot	pt
af23560	23560	23560	484256	1.096	0.324	0.33	0.208
cage11	39082	39082	559722	1.954	0.314	0.472	0.214
cage12	130228	130228	2032536	7.912	1.3	3.018	0.952
e30r2000	9661	9661	306356	0.946	0.362	0.262	0.24
e40r0100	17281	17281	553956	1.704	0.664	0.478	0.44
lhr10	10672	10672	232633	0.802	0.368	0.234	0.246
lhr14	14270	14270	307858	1.074	0.486	0.304	0.324
lhr34	35152	35152	764014	2.646	1.208	0.77	0.81
lhr71c	70304	70304	1528092	5.324	2.41	1.53	1.618
lpcrea	3516	7248	18168	0.292	0.022	0.038	0.012
lpcreb	9648	77137	260785	14.368	1.498	2.516	0.992
lpcred	8926	73948	246614	14.178	1.524	2.572	1.008
lpdfl001	6071	12230	35632	0.402	0.022	0.05	0.018
lpken11	14694	21349	49058	0.414	0.034	0.072	0.024
lpken13	28632	42659	97246	1.236	0.088	0.188	0.064
lpken18	105127	154699	358171	15.822	0.634	1.5	0.472
lpmarosr7	3136	9408	144848	0.786	0.206	0.186	0.138
lppds10	16558	49932	107605	0.582	0.048	0.12	0.034
lppds20	33874	108175	232647	1.354	0.112	0.258	0.086
lpstocfor3	16675	23541	76473	0.346	0.022	0.04	0.014

4.3.4 Comparison

Table 4.7 presents a comparison for running time for ColPack and DSJM toolkit. For comparison we are showing running time for Incidence Degree ordering in this table. On the left side, we list the name and structural properties of the matrices. On the right side, we list the ordering time(*ot*) and partitioning time(*pt*) for each software. We can see that ordering time is significantly larger than partitioning time in each case. Table 4.7 clearly shows that DSJM is efficient in terms of CPU cycles, as it requires less amount of time to perform the orderings. Figure 4.1 also compares the running time between them.

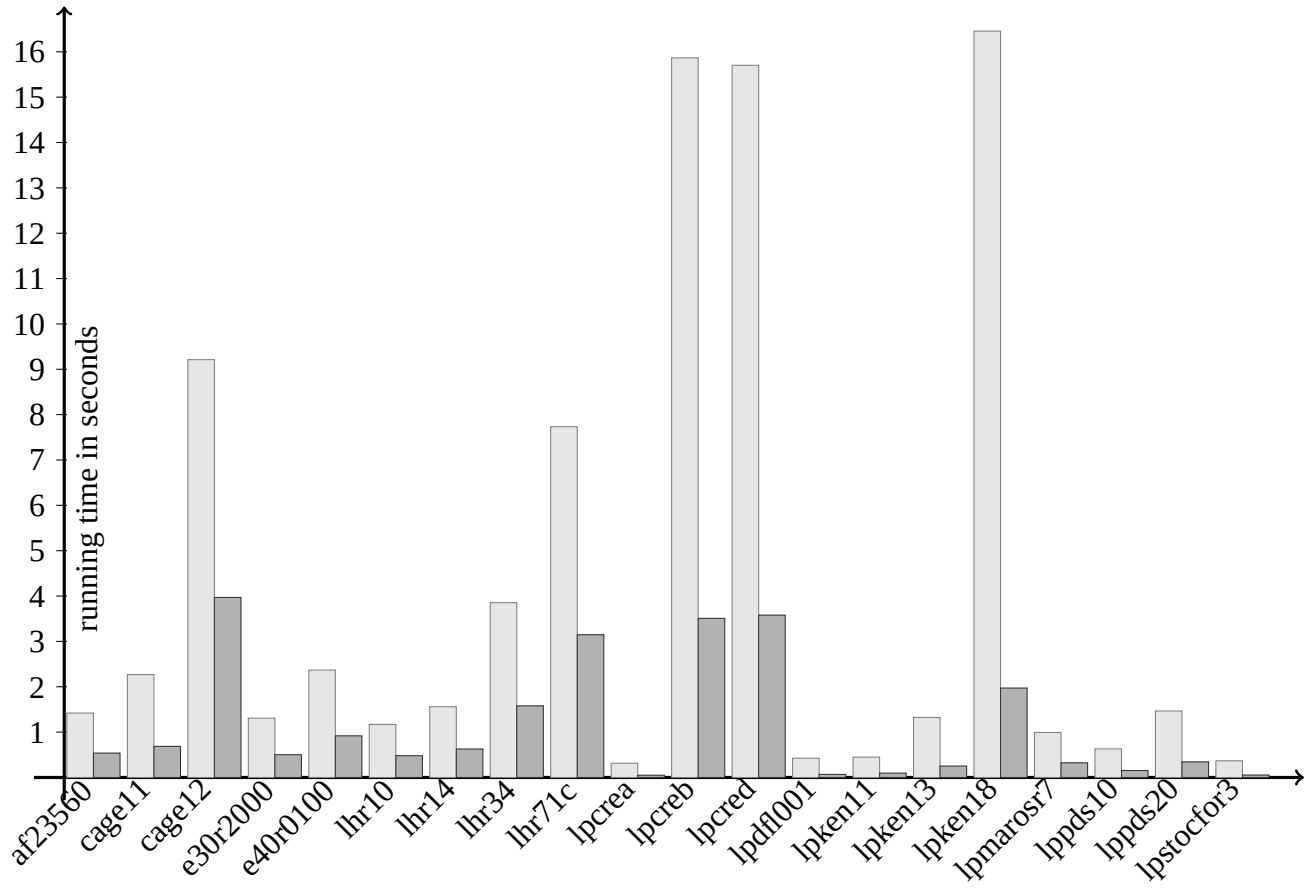

Figure 4.1: Comparison of Running time for IDO between ColPack and DSJM. Running time for Colpack is indicated by lighter shade, and darker shade bars represent DSJM.

In addition to efficient execution, DSJM is also partitioning the columns into less number of structurally orthogonal groups. Table 4.8 presents a comparison of the coloring results from both of the package. We list only the best partition for each of the package.

4.4 Hybrid Coloring

RLF's superior partitioning results comes with increased computational time, as seen earlier in this chapter. Hybrid coloring can be used as a parametrized version of RLF, which helps to trade off quality of partitioning for faster execution of RLF algorithm. Hybrid Col-

Table 4.8: Partitioning Results

Matrix Name	m	n	nnz	ρ_{max}	ColPack	DSJM
af23560	23560	23560	484256	21	41 (SLO)	37 (RLF)
cage11	39082	39082	559722	31	62 (SLO)	54 (RLF)
cage12	130228	130228	2032536	33	68 (SLO)	56 (RLF)
e30r2000	9661	9661	306356	62	68 (LFO)	65 (RLF)
e40r0100	17281	17281	553956	62	66 (LFO)	66 (LFO)
lhr10	10672	10672	232633	63	63 (SLO)	63 (SLO)
lhr14	14270	14270	307858	63	63 (SLO)	63 (RLF,SLO)
lhr34	35152	35152	764014	63	63 (SLO)	63 (RLF, SLO)
lhr71c	70304	70304	1528092	63	63 (SLO)	63 (RLF, SLO)
lpcrea	3516	7248	18168	360	360 (ALL)	360 (ALL)
lpcreb	9648	77137	260785	844	844 (IDO)	844 (RLF,IDO,LFO,SDO)
lpcred	8926	73948	246614	808	808 (ALL)	808 (ALL)
lpdfl001	6071	12230	35632	228	228 (ALL)	228 (ALL)
lpken11	14694	21349	49058	122	123 (IDO)	122 (RLF)
lpken13	28632	42659	97246	170	171 (IDO,SLO)	170 (RLF, IDO, SDO)
lpken18	105127	154699	358171	325	325 (SLO)	325 (RLF,SLO)
lpmarosr7	3136	9408	144848	48	70 (LFO)	76 (RLF)
lppds10	16558	49932	107605	96	96 (ALL)	96 (ALL)
lppds20	33874	108175	232647	96	96 (ALL)	96 (ALL)
lpstocfor3	16675	23541	76473	15	15 (ALL)	15 (ALL)
Total				3573	3693	3670

oring employs RLF and SLO to achieve better partitioning result while keeping running time low. SLO ordering was chosen as the accompanied ordering algorithm because:

1. SLO ordering is closer to RLF in performance with respect to partitioning.

2. SLO fits naturally with RLF since SLO and RLF produces the ordering at the opposite ends.

Table 4.9 lists number of colors and running time for hybrid RLF-SLO coloring. We list the number of colors obtained from RLF in column 2. If we parametrize RLF-SLO to process the first 10 percentage of vertices in SLO before switching to RLF, it partitions the columns in less time, but with usually higher number of structurally orthogonal column groups. Numerical observations for parametrized value $0.1 \equiv 10\%$ is given in column 3 and column 4. Similar results for parameter value 0.4 and 0.8 is given in the subsequent columns.

4.5 Summary

RLF clearly outperforms all other ordering algorithm in terms of number of structurally orthogonal partitions produced. RLF running time can be larger than running time of other ordering routines. In many cases, Jacobian matrices has to be estimated repeatedly, while the ordering can be done only once. So, spending more time in RLF to obtain less colors is justified in most cases. DSJM also performs faster in terms of running time for similar algorithms implemented by ColPack. The efficient execution can be attributed to the data structures used by DSJM, which uses flat array data structure, thus utilizing hierarchical memory architecture.

Table 4.9: Number of Colors and Required time in seconds for RLF-SLO , with RLF running over first $10, 40, 80$ percentage of vertices.

Matrix	RLF	0.1		0.4		0.8	
	Color	Color	Time	Color	Time	Color	Time
af23560	37	41	1.858	40	4.246	37	5.486
cage11	54	62	2.64	60	5.686	59	8.526
cage12	56	67	14.614	65	34.678	63	52.238
e30r2000	65	68	1.988	68	4.88	66	6.042
e40r0100	67	70	3.718	69	9.068	67	11.154
lhr10	64	64	0.814	64	1.404	64	1.626
lhr14	63	63	1.09	63	1.844	63	2.168
lhr34	63	63	2.698	63	4.556	63	5.402
lhr71c	63	63	5.376	63	9.11	63	10.814
lpcrea	360	360	0.106	360	0.118	360	0.378
lpcreb	844	845	18.076	845	122.022	844	229.046
lpcred	808	808	20.682	808	135.942	808	246.188
lpdfl001	228	228	0.144	228	0.202	228	0.392
lpken11	122	123	0.186	122	0.208	122	0.45
lpken13	170	171	0.476	170	0.528	170	1.216
lpken18	325	326	3.2	325	3.456	325	12.15
lpmarosr7	76	85	0.62	88	1.092	81	3.608
lppds10	96	96	0.338	96	0.552	96	0.754
lppds20	96	96	0.798	96	1.332	96	1.796
lpstocfor3	15	15	0.102	15	0.122	16	0.17

Chapter 5

Conclusion and Future Work

5.1 Conclusion

In this thesis we have studied methods for estimation of Sparse Jacobian matrices. We felt that there has been a gap for a modern tool for estimation of sparse Jacobian matrices since DSM's [7] release in 1984. This thesis has been an effort to provide a modern software toolkit for estimating Jacobian matrices. We have provided well known algorithms along with some new ones for column partitioning problem. Though RLF has been used previously for graph coloring, we have implemented it for column partitioning problem for the first time. We have found that RLF produces better results than other widely used heuristics for column partitioning problem. Our implementation has tried to exploit the data structures used for sparse matrices. We have seen that the software toolkit proved to be competitive in both running time and number of partitions achieved. We provided C++, C and MATLAB interfaces for the algorithms for better integration with existing applications. We hope that it will be widely adopted by both practitioners and researchers.

5.2 Future Research Direction

1. We want to extend the algorithms for Column Segmented matrix [21]. A column segmented matrix can be partitioned without explicitly constructing it. Moreover, the number of groups for a column segmented matrix will not be any larger than the partitions achieved for the original matrix. This work is in progress.

2. We would like to extend the toolkit for distributed computing environment, we have been looking into Condor[6] to exploit idle CPU power typically available to academic and corporate settings to solve column partitioning problem for large instances.

A distributed computing environment allows us to run different branches of a partitioning problem in different machines simultaneously. The heuristics can be re-implemented with minimal communication overheads between the running instances to minimize turnover time.

Bibliography

[1] URL: http://math.nist.gov/MatrixMarket/collections/hb.html. [Online; accessed July-2009].

[2] URL: http://math.nist.gov/MatrixMarket/matrices.html. [Online; accessed July-2009].

[3] URL: http://www.netlib.org/lp/data/. [Online; accessed July-2009].

[4] A. Bouaricha and R.B. Schnabel. Tensor methods for large sparse systems of nonlinear equations. *Mathematical programming*, 82(3):377–400, 1998.

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

[6] A. Bricker, M. Litzkow, and M. Livny. Condor technical summary. Technical report, Citeseer, 1991.

[7] T.F. Coleman, B.S. Garbow, and J.J. More. Software for estimating sparse Jacobian matrices. *ACM Transactions on Mathematical Software (TOMS)*, 10(3):329–345, 1984.

[8] 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.

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

[10] T. Davis. University of Florida sparse matrix collection. *NA Digest*, 97(23):7, 1997. [Online; accessed July-2009].

[11] J.E. Dennis and R.B. Schnabel. *Numerical methods for unconstrained optimization and nonlinear equations*. Society for Industrial Mathematics, 1996.

[12] Thomas Ericsson. URL: http://www.math.chalmers.se/ thomas/PDC/springer.pdf. `http://www.math.chalmers.se/~thomas/PDC/springer.pdf`. [Online; accessed July-2009].

[13] S.A. Forth. An efficient overloaded implementation of forward mode automatic differentiation in matlab. *ACM Transactions on Mathematical Software (TOMS)*, 32(2):195–222, 2006.

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

[15] M.R. Garey and D.S. Johnson. *Computers and intractability*. Freeman San Francisco, 1979.

[16] A.H. Gebremedhin, F. Manne, and A. Pothen. What color is your Jacobian? Graph coloring for computing derivatives. *SIAM REVIEW*, 47(4):629, 2005.

[17] AH Gebremedhin, A. Tarafdar, D. Nguyen, and A. Pothen. ColPack.

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

[19] A. Griewank and A. Walther. *Evaluating derivatives: principles and techniques of algorithmic differentiation*. Society for Industrial and Applied Mathematics (SIAM), 2008.

[20] M. Hasan, S. Hossain, and T. Steihaug. DSJM: A Software Toolkit for Direct Determination of Sparse Jacobian Matrices. *SIAM workshop on Combinatorial Scientific Computing*, 2009.

[21] S. Hossain. Cseggraph: a graph colouring instance generator. *International Journal of Computer Mathematics*, 86(10):1956–1967, 2009.

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

[23] F.T. Leighton. A graph coloring algorithm for large scheduling problems. *Journal of Research of the National Bureau of Standards*, 84(6):489–503, 1979.

[24] E. Malaguti and P. Toth. A survey on vertex coloring problems. *International Transactions in Operational Research*, 2009.

[25] M. Market. URL: http://math. nist. gov. `http://math.nist.gov/`. [Online; accessed July-2009].

[26] Y. Saad and Y. Saad. *Iterative methods for sparse linear systems*. PWS Pub. Co., 1996.

Appendix A

Compilation and Usage

A.1 Use Case Scenario

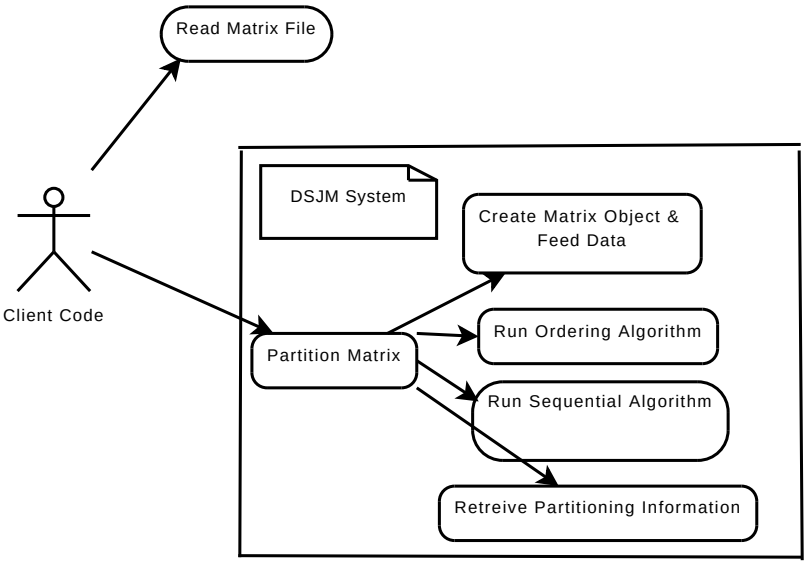

Figure A.1: Use Case of DSJM Software Toolkit

DSJM[20] software toolkit is used to find a structurally orthogonal partition of large sparse matrices. The client code will read sparsity information of the matrix and construct a Matrix object provided by DSJM. The following steps describe a use case scenario for the client code:

1. Creates a *Matrix* object provided by *DSJM*.
2. Reads the row and column indices for each non-zero element of the sparse matrix, and construct a *Matrix* object.
3. Run an ordering and greedy coloring method provided by the *DSJM* toolkit to obtain a structurally orthogonal partition.

A.2 Compilation

In a C++ settings, one can include our source code in her compilation unit, and directly use the *Matrix* class. Alternatively, the toolkit can be accessed as a statically linked library. We provide a *Makefile* based build process to obtain the statically linked library.

You can compile the source code of *DSJM* toolkit with the following commands given in the top directory of DSJM source distribution:

```
./configure
make
```

The compiled library is a statically linkable file **libmatrix.a** and can be found on the **src** directory. You can link against this library for compiling your application.

A.2.1 Linking against libmartix.a

Assuming you are using *g++* compiler,you can link your application against *libmatrix.a* with the following way:

```
$ g++ your_application.cpp libmatrix.a -i /path/to/dsjm/source
```

A.3 User Interface

Features provided by DSJM toolkit has been exposed through one C++ key class, named *Matrix*.

In this section we will describe how to use the *Matrix* class.

We assume the client code has at least the following information about the target sparse matrix:

1. *M*, number of rows in the matrix.
2. *N*, number of columns in the matrix.
3. *nnz*, number of nonzero elements in the matrix.

To get the functionality of DSJM we have to create an object of the *Matrix* class.

```
Matrix matrix(M,N,nnz,false);
```

After creating the object we have to provide the sparsity pattern (row number and column number) to the matrix object.

Note A.3.1 *The indices in the Matrix object are counted from* 1, *not from* 0.

Then we call four preprocessing functions on the matrix object,and the matrix data structure will be ready for ordering or coloring algorithms to run. `computeCCS` method constructs Compressed Column Storage from the sparsity pattern. `compress` method finds duplicate entries, and discards them. `computeCRS` method constructs a Compressed Row Storage, and `computeDegree` method computes the degree information in the intersection graph $G(A)$.

```
matrix.computeCCS();
int nnz = matrix.compress();
matrix.computeCRS();
matrix.computedegree();
```

To run any of the *slo*,*lfo* or *ido* ordering methods we have to call two separate functions, one is the desired ordering function, and then we have to call the greedy partitioning function *greedycolor*.

As an example, for *lfo* ordering, we have to execute the following instructions:

```
int *order = new int[N+1];
matrix.lfo(order);
int *color = new int[N+1];
int maxgrp = matrix.greedycolor(order,color);
```

But *rlf* and *sdo* orderings method have the partitioning algorithm built in, so for partitioning the columns through RLF and SLO heuristics we **must not** call `greedycolor()` method. For example, RLF we do the following

```
int *color = new int[N+1];
int maxgrp = matrix.rlf(color);
// Don't call matrix.greedycolor() after rlf.
```

By this point, every column is assigned to one of the structurally orthogonal groups which are numbered from 1 to maxgrp and the related group for each column is stored in the `color` array such that color[i] represents the group number of column *i*.

A.3.1 Example Usage of Matrix Object

```
Matrix matrix(M,N,nnz, false);

for(int i = 1 ; i <= nnz; i++)
{
    int row, column;
    readNextNonzeroLocation(&row,&column);
    // Client Code supplied Method
    // ,may also be supplied from
    // an array

    matrix.entry(row,col);
}
```

```
matrix.computeCCS();
int nnz = matrix.compress();
matrix.computeCRS();
matrix.computedegree();

int *order = new int[N+1];
matrix.lfo(order);

int *color = new int[N+1];
int maxgrp = matrix.greedycolor(order,color);
// Don't use this function for RLF

for (int j = 1; j <= N; j++)
{
    printf("Column J belongs to %d partition\n",color[i]);
}
```

A.4 Matrix Class

Following functions are available to the user of `Matrix` class.

A.4.1 `Matrix(int M, int N, int nz, bool values)`

Constructor of the class. The parameters represent the number of rows, number of columns, and number of nonzero values in the matrix. If the fourth parameter, `values` is true then the matrix object stores values of the nonzero items. Otherwise, it only stores the sparsity pattern and disregards the original values.

A.4.2 `bool computeCCS()`

Purpose Computes *Compressed Column Storage(CCS)* format of the sparse matrix. The CCS format stores the columns of matrix *A* in three member arrays in Matrix object: `<id:jpntr>`, `<id:indRow>` and `<id:x>`. Data member `<id:x>` is empty if `<id:value>`, a boolean member variable evaluates to false.

Pre-condition Assumes that the matrix definition is stored in co-ordinate format in `<id:indRow>` and `<id:indCol>` integer array. For every non-zero position in the sparse matrix there is two entry: `indRow[i]` and `indCol[i]` holding the *row*, and *column* coordinate of the nonzero entry. If `<id:value>` is true then `x[i]` stores the corresponding nonzero item.

Post-condition Column-oriented definition of the sparse matrix is stored in the two array

<id:jpntr> and <id:indRow>. If value of the nonzero items are being stored , then <id:x> is also organized in column oriented definition.

Return value Returns `true` when the function is executed successfully, otherwise returns `false`.

A.4.3 `bool computeCRS()`

Purpose Computes *Compressed Row Storage(CRS)* definition of the sparse matrix. The CRS format stores the rows of matrix A in two member arrays in Matrix object: <id:ipntr>, and <id:indCol>. Value array <id:x> is not stored in row oriented definition.

Pre-condition Assumes that the matrix definition is stored in CCS format in <id:indRow> and <id:jpntr> integer array and duplicate entries has been removed by calling `computeCCS()` method and `compress()` method.

Post-condition Row-oriented definition of the sparse matrix is stored in the two array <id:ipntr> and <id:indCol>.

Return value Returns `true` when the function is executed successfully, otherwise returns `false`.

A.4.4 `int compress()`

Purpose Removes duplicate entries from the column-oriented definition of the sparse matrix, and compresses the member arrays <id:indRow>, <id:jpntr> and <id:x> array.

Pre-condition Assumes that the sparsity pattern has been stored in column-oriented definition in <id:jpntr>, <id:indRow> and <id:x> array by calling `computeCCS()` method.

Post-condition Removes duplicate entry and reorganizes <id:indRow>, <id:jpntr> and <id:x> array.

Return value Returns number of unique nonzero items when the function is executed successfully, otherwise returns zero.

A.4.5 `bool computedegree()`

Purpose Given the sparsity pattern of a matrix A, this method determines the degree sequence of the sparse matrix A (of the vertices of the column intersection Graph $G(A)$).

Pre-Condition The matrix object is nonempty. Assumes that the `computeCCS()`, `compress()` and `computeCRS()` has been called prior calling this function, such that matrix object holds the sparsity pattern in Compressed Column and Compressed Row storage format.

Post-Condition Degree information for the columns of matrix A(graph $G(A)$) is stored in the data member `<id:ndeg>`, an integer array of size $n+1$, such that if $k =$ `ndeg[j]` then the column j has degree k, where $j = 1, 2, \ldots, n$.

Return value Returns `true` when the function is executed successfully, otherwise returns `false`.

A.4.6 `bool slo(int *order)`

Purpose Computes *Smallest-Last Ordering (SLO)* of the columns of a sparse matrix A (i.e. the vertices of the column intersection graph $G(A)$).

Pre-condition The matrix object is nonempty. Assumes that the degree of the columns have already been computed in the data member `<id:ndeg>`, an integer array of size $n+1$, using `computedegree()` method.

Post-condition The SLO ordering of matrix A (graph $G(A)$) is stored in the out-parameter `<id:order>`, an integer array of size $n+1$, such that if $k =$ `order[j]` then the column j is the k-th element, $k = 1, 2, \ldots, n$, in the SLO ordering, and $j = 1, 2, \ldots, n$.

Parameters Out-parameter `<id:order>`, an integer pointer to an array of size $n+1$. The array will contain the ordering information when the function normally returns.

Return value Returns `true` when the function is executed successfully, otherwise returns `false`.

A.4.7 `bool ido(int *order)`

Purpose Computes *Incidence-Degree Ordering (IDO)* of the columns of a sparse matrix A (i.e. the vertices of the column intersection graph $G(A)$).

Pre-condition The matrix object is nonempty. Assumes that the degree of the columns have already been computed in the data member `<id:ndeg>` integer array of size $n+1$ using `computeDegree()` method.

Post-condition The *IDO* ordering of matrix A (graph $G(A)$) is stored in the out-parameter `<id:order>`, an integer array of size $n+1$, such that if $k =$ `order[j]` then the column j is the k-th element, $k = 1, 2, \ldots, n$, in the IDO ordering, and $j = 1, 2, \ldots, n$.

Parameters Out-parameter `<id:order>`, an integer pointer to an array of size $n+1$. The array will contain the ordering information when the function normally returns.

Return value Returns `true` when the function is executed successfully, otherwise returns `false`.

A.4.8 `bool lfo(int *order)`

Purpose Computes *Largest-First Ordering (LFO)* of the columns of a sparse matrix A (i.e. the vertices of the column intersection graph $G(A)$).

Pre-condition The matrix object is nonempty. Assumes that the degree of the columns have already been computed in the data member `<id:ndeg>`, an integer array of size $n+1$, using `computeDegree()` method.

Post-condition The *LFO* ordering of matrix A (graph $G(A)$) is stored in the out-parameter `<id:order>`, an integer array of size $n+1$, such that if $k =$ `order[j]` then the column j is the k-th element, $k = 1,2,\ldots,n$, in the LFO ordering, and $j = 1,2,\ldots,n$.

Parameters Out-parameter `<id:order>`, an integer pointer to an array of size $n+1$. The array will contain the ordering information when the function normally returns.

Return value Returns `true` when the function is executed successfully, otherwise returns `false`.

A.4.9 `int sdo(int *color)`

Purpose Computes *Saturation-Degree Coloring*(SDO) of the columns of a sparse matrix A (i.e. the vertices of the column intersection graph $G(A)$).

Pre-condition The matrix object is nonempty. Assumes that the degree of the columns have already been computed in the data member `<id:ndeg>`, an integer array of size $n+1$, using `computeDegree()` method.

Post-condition *SDO* coloring of Matrix A (graph $G(A)$) is stored in the in-out-parameter `<id:color>`, an integer array of size $n+1$, such that if $k =$ `color[j]` then the column j is colored with color k, where $j = 1,2,\ldots,n$.

Parameters Out-parameter `<id:color>`, an integer pointer to an array of size $n+1$. The array will contain the color values of the columns in successful completion. The integer array uses 1-based indexing.

Return value Returns the number of colors if succeeds, otherwise returns 0 (zero).

A.4.10 `int greedycolor (int *order, int *color)`

Purpose Computes the greedy coloring of the columns of a sparse matrix A (i.e. the vertices of the column intersection graph $G(A)$).

Pre-condition The matrix object is nonempty. Assumes that an ordering has been provided in the in-parameter `<id:order>`, an integer array of size $n+1$, such that `order[1]`, ..., `order[n]` is a permutation of $\{1,...,n\}$.

Post-condition The greedy coloring of Matrix A (graph $G(A)$) is stored in the in-out-parameter `<id:color>`, an integer array of size $n+1$, such that if $k =$ `color[j]` then the column j is colored with color k, where $j = 1,2,\ldots,n$.

Parameters In-parameter `<id:order>`, an integer pointer to an array of size $n+1$, containing a permutation of $\{1,\ldots,n\}$. The integer array uses 1-based indexing.

In-out-parameter `<id:color>`, an integer pointer to an array of size $n+1$, it stores the color values of the columns in successful completion.The integer array uses 1-based indexing.

Return value Returns the number of colors if succeeds, otherwise returns 0 (zero).

A.4.11 `int rlf(int *color)`

Purpose Computes *Recursive Largest-First* coloring (RLF) of the columns of a sparse matrix A (i.e. the vertices of the column intersection graph $G(A)$).

Pre-condition The matrix object is nonempty. Assumes that the degree of the columns have already been computed in the data member `<id:ndeg>`, an integer array of size $n+1$, using `computeDegree()` method.

Post-condition RLF coloring of Matrix A (graph $G(A)$) is stored in the in-out-parameter `<id:color>`, an integer array of size $n+1$, such that if $k =$ `color[j]` then the column j is colored with color k, where $j = 1,2,\ldots,n$.

Parameters Out-parameter `<id:color>`, an integer pointer to an array of size $n+1$. The array will contain the color values of the columns in successful completion. The integer array uses 1-based indexing.

Return value Returns the number of colors if succeeds, otherwise returns 0(zero).

A.4.12 `void rlf_slo(int *ngrp, int p)`

Purpose Computes RLF and SLO coloring (Hybrid Coloring) of the columns of a sparse matrix A (i.e. the vertices of the column intersection graph $G(A)$), partitions first p columns according to *RLF* ordering and then colors remaining columns with *SLO* ordering algorithm.

Pre-condition The matrix object is nonempty. Assumes that the degree of the columns have already been computed in the data member `<id:ndeg>`, an integer array of size $n+1$, using `computeDegree()` method.

Post-condition RLF-SLO coloring of Matrix A (graph $G(A)$) is stored in the in-out-parameter `<id:color>`, an integer array of size $n+1$, such that if k = `color[j]` then the column j is colored with color k, where $j = 1, 2, \ldots, n$.

Parameters Out-parameter `<id:color>`, an integer pointer to an array of size $n+1$. The array will contain the color values of the columns in successful completion. The integer array uses 1-based indexing.

Return value `void`.

A.5 Reading Matrix Market Data File

DSJM code depends on the application's code to supply the data of the sparse matrix. We also provide a way to read *Matrix Market* exchange format.

A.5.1 Reading Matrix Market Banner

Matrix Market format provides a banner which lists meta-data for the matrix. The following code snippet can retrieve important meta-data by reading .mtx file.

```
MM_typecode matcode;
int ret_code;

FILE *f;
f  = fopen("filename.mtx", "r");
if (mm_read_banner (f, &matcode) != 0)
  {
    fprintf (stderr,
    "filename.mtx -> Could not process Matrix Market banner.\n");
    exit (1);
  }

/**
 * ----------------------------------------------------------------
 * This is how one can screen matrix types if their applicaiton
 * only supports a subset of the Matrix Market data types.
 **/

if (mm_is_complex (matcode) && mm_is_matrix (matcode) &&
    mm_is_sparse (matcode))
  {
    printf ("Sorry, this application does not support ");
    printf ("Market Market type: [%s]\n", mm_typecode_to_str (matcode));
```

```
    exit (1);
  }

/**
 *
 * Find out the size of the sparse matrix
 **/

if ((ret_code = mm_read_mtx_crd_size (f, &M, &N, &nz)) != 0)
  exit (1);


is_symmetric = mm_is_symmetric(matcode);
is_pattern = mm_is_pattern(matcode);
nz = 2 * nz;
```

A.5.2 Reading sparsity pattern

The client code can provide the data to the `matrix` object using the following code.

```
// As we are not going to use the value, we are simply using
// this placeholder variable 'value' to read each line
// from the input matrix.


    double value;

    for ( int i = 1,row,col ; i <= nz; i++ )
    {
        if (is_pattern)
        {
            fscanf (f, "%d %d\n", &row, &col);
        }
        else
        {
            fscanf (f, "%d %d %lg\n", &row, &col, &value);
        }

        matrix->setIndRowEntry(i,row);
        matrix->setIndColEntry(i,col);
```

```
        if(is_symmetric)
        {
            matrix->setIndRowEntry(i + nz,col);
            matrix->setIndColEntry(i + nz,row);
        }

    }

    if (f != stdin)
        fclose(f);
```

A.6 Matlab Usage

The functionalities of the *DSJM* toolkit has been exposed through the `dsjmcolor()` function. This function requires two parameters, a MATLAB sparse matrix and a method name. For example, to obtain *Smallest-Last-Coloring* on a matrix `A` in MATLAB we would have to call the function in the following way:

```
B = dsjmcolor(A,'slo');
```

The coloring assignment will be stored in the `B` matrix, i.e., *i*th column will be in the orthogonal column group `B(i)`, where $i \in \{1,\ldots,n\}$.

The following functions of the *DSJM* can be called through `dsjmcolor()`:

1. Matrix Class
    - Largest First Ordering Coloring.

      ```
      B = dsjmcolor(A,'lfo');
      ```
    - Smallest Last Ordering Coloring.

      ```
      B = dsjmcolor(A,'slo');
      ```
    - Incidence Degree Coloring.

      ```
      B = dsjmcolor(A,'ido');
      ```
    - Saturation Degree Coloring.

      ```
      B = dsjmcolor(A,'sdo');
      ```
    - Recursive Largest First Coloring.

      ```
      B = dsjmcolor(A,'rlf');
      ```

DSJM functionalities are exposed to *MATLAB* through *MEX*(MATLAB executable) interface.

A.6.1 Compiling for Matlab

MEX source codes are located in `mex` directory in source distribution. To compile the mex files, you have to perform the following steps :

1. Compile and build static library `libmatrix.a`. See section (A.2) for details on compilation of the static library.
2. Edit `mex/Makefile` such that:
   (a) MATLABHOME contains the Matlab installation path.
   ```
   MATLABHOME = /path/to/matlab/installation
   ```
   (b) MEX variable contains the full path-name of the `mex` executable.
   ```
   MEX        = /path/to/mex/executable
   ```
3. Run `make` in `mex` directory to compile the `*.mexglh` files.
   ```
   $ make
   ```

A.6.2 Calling Matrix functions from Matlab

Setup Path

Before calling *DSJM* functions from MATLAB , make sure that `mex` directory is added to the Matlab search path. You can type the following command in the `Matlab` console so that MATLAB is setup correctly to find *DSJM* mex files.

```
>> addpath('/path/to/mex/directory')
```