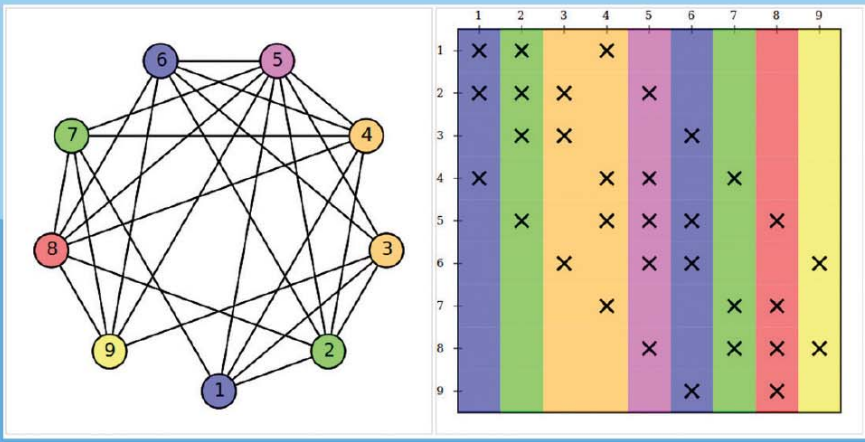
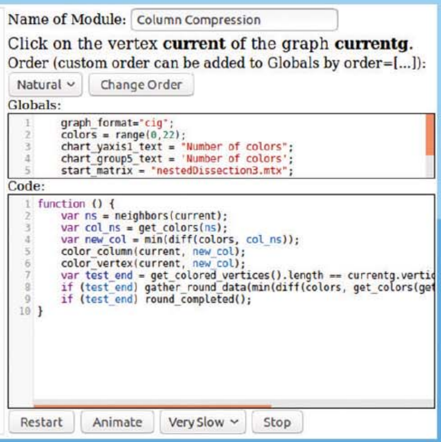
# Combining partial Jacobian computation and preconditioning: New heuristics, educational modules, and applications

Mohammad Ali Rostami





Name of Module: Column Compression

Click on the vertex **current** of the graph **currentg**.

Order (custom order can be added to Globals by order=[...]):

Globals:

```
    graph_format="cig";
    colors = range(0,22);
    chart_yaxis1_text = "Number of colors";
    chart_group5_text = 'Number of colors';
    start_matrix = "nestedDissection3.mtx";
```

Code:

```
function () {
    var ns = neighbors(current);
    var col_ns = get_colors(ns);
    var new_col = min(diff(colors, col_ns));
    color_column(current, new_col);
    color_vertex(current, new_col);
    var test_end = get_colored_vertices().length == currentg.vertic
    if (test_end) gather_round_data(min(diff(colors, get_colors(get
    if (test_end) round_completed();
}
```

Cuvillier Verlag Göttingen
Internationaler wissenschaftlicher Fachverlag

Combining partial Jacobian computation and preconditioning:
New heuristics, educational moduls, and applications

seit 1558

# Combining partial Jacobian computation and preconditioning: New heuristics, educational modules, and applications

Mohammad Ali Rostami
Dissertation at Faculty of Mathematics and Computer Science
Friedrich Schiller University Jena, Germany

Reviewers:

Prof. Martin Bücker, FSU Jena, Germany
Prof. Trond Steihaug, University of Bergen, Norway

Defense: 27. September 2017

**Bibliografische Information der Deutschen Nationalbibliothek**
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.
1. Aufl. - Göttingen: Cuvillier, 2017
Zugl.: Friedrich-Schiller-Universität Jena, Univ., Diss., 2017

© CUVILLIER VERLAG, Göttingen 2017
Nonnenstieg 8, 37075 Göttingen
Telefon: 0551-54724-0
Telefax: 0551-54724-21
www.cuvillier.de

1. Auflage, 2017
Gedruckt auf umweltfreundlichem, säurefreiem Papier aus nachhaltiger Forstwirtschaft.

ISBN 978-3-7369-9636-6
eISBN 978-3-7369-8636-7

# Abstract

Solving problems originating from real-world applications is often based on the solution of a system of linear equations whose coefficient matrix is a large sparse Jacobian matrix. Hence, there is research to exploit the sparsity structure and to decrease the amount of storage. In contrast to full Jacobian computation in which all nonzero elements are to be determined, partial Jacobian computation is looking at a subset of these elements. Partial Jacobian computation can therefore be faster and more efficient than full Jacobian computation. Since Jacobian matrix-vector products are needed in iterative solvers, these types of linear systems can be efficiently solved using automatic differentiation. Determining these nonzero elements in full or partial Jacobian computations by automatic differentiation techniques can be modeled as graph coloring in the language of graph theory. On the other hand, preconditioning techniques are used to improve the convergence of iterative solvers and typically need access to all nonzero elements of the Jacobian matrix. So, a sparsification is applied to the Jacobian matrix before computing the preconditioner. The nonzero elements obtained from the sparsification are considered as the required elements in a restricted coloring. Lülfesmann (PhD thesis, RWTH Aachen University, 2012) introduced a procedure that selects a subset of the remaining nonrequired elements. The approach then adds this subset to the sparsified matrix such that neither fill-in is created nor an increase in the number of colors happens. This thesis consists of two parts. In the first part, we look at different ways to optimize the process of selecting these nonrequired elements. We introduce three new coloring heuristics and compare them with each other as well as with existing approaches. Also, we look at a particular case in which we consider only the diagonal elements as required elements. For this case, we generalize a previous result and introduce a new coloring heuristic. To evaluate our proposed heuristics in practice, we apply them to a problem from geoscience. Finally, we describe our new software package implementing these new heuristics. In the second part of this thesis, we introduce a collection of interactive educational modules to teach not only graph coloring, but also other concepts from combinatorial scientific computing in the classroom. These modules are designed to involve the students more thoroughly in the process of learning. At the end, we explain the design of this collection and outline its implementation.

# Zusammenfassung

Die Lösung von realistischen Anwendungsproblemen basiert oft auf der Lösung von linearen Gleichungssystemen, deren Koeffizientenmatrizen große dünnbesetze Jacobi-Matrizen sind. Deshalb gibt es eine Fülle von Forschungsarbeiten mit dem Ziel, die Struktur dieser Matrizen zur Reduktion von Rechenzeit und Speicher auszunutzen. Im Gegensatz zu der vollständigen Berechnung von Jacobi-Matrizen, in der alle Nichtnullelemente bestimmt werden müssen, wird in einer partiellen Berechnung lediglich eine Teilmenge aller Nichtnullelemente bestimmt. Die partielle Berechnung kann daher im Vergleich zu einer vollständigen Berechnung zu schnelleren und effizienteren Verfahren führen. Da Jacobi-Matrix-Vektor-Produkte in iterativen Lösern benötigt werden, lassen sich diese Typen von linearen Systemen mit Techniken des automatischen Differenzierens effizient lösen. Die Bestimmung von Nichtnullelementen durch vollständige oder partielle Berechnungen von Jacobi-Matrizen kann in der Sprache der Graphentheorie als Problem einer Graphenfärbung modelliert werden. Auf der anderen Seite werden Vorkonditionierungstechniken verwendet, um die Konvergenz von iterativen Lösern zu verbessern. Solche Vorkonditionierungstechniken greifen meist auf alle Nichtnullelemente der Jacobi-Matrix zu. Daher kann es vorteilhaft sein, eine Ausdünnung auf die Jacobi-Matrix anzuwenden, bevor die Vorkonditionierung berechnet wird. Die nach der Ausdünnung verbleibenden Nichtnullelemente werden als benötigte Elemente bezeichnet und in einer eingeschränkten Färbung verwendet. Lülfesmann (Dissertation, RWTH Aachen, 2012) hat ein Verfahren eingeführt, das eine Teilmenge der nichtbenötigten Elemente auswählt und zu der ausgedünnten Jacobi-Matrix hinzufügt, so dass weder Füllelemente noch eine Erhöhung der Anzahl der Farben erfolgt. Die nun vorliegende Dissertation besteht aus zwei Teilen. Im ersten Teil betrachten wir verschiedene Möglichkeiten, den Prozess der Auswahl dieser nichtbenötigten Elemente zu optimieren. Wir stellen drei neue Färbungsheuristiken vor und vergleichen sie sowohl untereinander als auch mit bisherigen Ansätzen. Wir betrachten auch einen Spezialfall, in dem als benötigte Elemente nur die Diagonalelemente zu bestimmen sind. Wir verallgemeinern dabei ein vorheriges Ergebnis und führen eine neue Färbungsheuristik ein. Um unsere vorgeschlagenen Heuristiken in der Praxis zu evaluieren, wenden wir sie auf eine Problemstellung aus den Geowissenschaften an. Schließlich betrachten wir unser neues Softwarepaket, das die neuen Heuristiken implementiert. Im zweiten Teil dieser Arbeit stellen wir eine Sammlung von interaktiven Lehrmodulen vor, die nicht nur Graphenfärbung, sondern auch andere Konzepte aus dem kombinatorischen wissenschaftlichen Rechnen im Klassenzimmer illustrieren. Diese Module wurden entwickelt, um die Studierenden noch gründlicher in den Prozess des Lernens zu involvieren. Schließlich erläutern wir den Entwurf dieser Sammlung von Lehrmodulen und beschreiben deren Implementierung.

# Acknowledgments

This project would not have been possible without the support of many people. I must express my first gratitude towards my supervisor *Prof. Dr. Martin Bücker* for his continuous support during my research, patience, immense knowledge, and attention to details. I should confess that I can not imagine a better and friendlier supervisor. *Prof. Dr. Trond Steihaug* was a good listener to my ideas that we discussed at the Oxford conference on algorithmic differentiation. I like to thank him for accepting to review my thesis. Besides, I would like to thank *Michael Lülfesmann* for his comments on my work. Finally, I wish to thank my wife *Masoumeh Seydi* and my best friend *Azin Azadi* for encouraging me throughout all my studies at university.

# Contents

# 1 Introduction

In scientific computing, it is common to solve a given problem by using methods from combinatorial mathematics. In particular, graphs are ubiquitous in numerical linear algebra if the underlying matrices are sparse. This thesis consists of two parts. In the first part, we discuss new coloring heuristics for the partial Jacobian computation. In the second part, we introduce a set of interactive educational modules teaching these graph problems in classroom.

This thesis is concerned with the solution of linear equations whose coefficient matrices are sparse, large, and nonsingular. Furthermore, it is assumed throughout this thesis that the coefficient matrix is a Jacobian matrix of some mathematical function. This Jacobian matrix is computed by Automatic Differentiation (AD) [1, 2] without truncation error. AD computes a product of a Jacobian and another matrix which is called the seed matrix. A careful choice of the seed matrix will reduce the computational effort as well as the storage. Different choices can be formulated as different graph problems.

These systems of linear equations are solved using iterative methods which are in practice accelerated by preconditioning techniques. Most preconditioning techniques need access to all nonzero elements of the Jacobian matrix which can lead to performance problems when standard techniques of AD are employed. Therefore, we consider an alternative approach that first applies a sparsification operator to the Jacobian matrix and then uses standard preconditioning techniques for the sparsified matrix. This process becomes complete by adding more nonzero elements to the sparsified matrix without producing more fill-in elements and without increasing efforts to set up the resulting sparsified matrix. We introduce new heuristics targeting the careful choice of these extra nonzero elements.

Our work is based on the idea of exploiting the sparsity pattern of a matrix in favor of reducing the computational efforts. The assumption here is that the sparsity pattern is known a priori, for example, from the formulation of a physical problem. Curtis, Powell, and Reid [3] were first to study the determination of a sparse Jacobian matrix based on the sparsity exploitation. Coleman and Moré [4] transformed this sparse matrix problem into the problem of vertex coloring in graph theory. The idea is to formulate the compression of the columns (similarly for rows) such that all the nonzero elements of the Jacobian matrix are determined. This is called unidirectional compression. There are other studies like [5] which consider the byproducts of some column compression schemes. Later, a bidirectional compression is introduced and analyzed in [6] and [7] which can result in larger savings in computation and storage. Several other graph models are studied further in Hossain and Steihaug [8, 9] like the pattern graph which keeps the structure of the matrix explicitly. Also, a recent graph formulation is to group together rows into blocks and partition the resulting column segments in [10]. On the other hand, rather than computing all nonzero elements, some other problems from scientific computing target the computation of only a proper subset of the nonzero elements. These problems are studied under the term of partial Jacobian computation. Gebremedhin, Manne, and Pothen [11] introduced the rules of the partial Jacobian computation and its corresponding graph problems. Several examples in the partial Jacobian computation are studied later in [12] and [13].

Lülfesmann [14] introduced for the first time the idea of combining partial Jacobian computation and ILU preconditioning. This idea defines a sparsification operator $\rho$ which is applied to the Jacobian matrix before the ILU preconditioner is computed. The nonzero elements selected by $\rho$ are called required elements. The remaining elements are called nonrequired elements. Lülfesmann [14] computes the seed matrix for automatic differentiation by considering a graph coloring restricted to these required elements. Then, a subset of nonrequired elements is added to the set of required elements such that the number of colors does not increase and no extra fill-in elements are generated in the ILU preconditioning. These elements are called the additionally required elements. In this thesis, we extend this idea further as follows. First, we define new coloring heuristics (both for distance-2 coloring and star bicoloring) to increase the number of additionally required elements without having a high increase in the number of colors. Then, we apply these new heuristics to an example from geoscience similar to an application from aerodynamics [15]. Later, we generalize a previous result from Lülfesmann [14] for the coloring restricted to diagonal elements. Finally, we introduce a software package to implement these new heuristics.

In the second part of this thesis, we summarize our previous publications [16, 17, 18, 19, 20] as well as discuss some new features to teach the coloring heuristics in classroom. In this part, we develop a collection of educational modules for teaching purposes. Each module illustrates side by side the matrix and graph view of a problem in scientific computing and its equivalent combinatorial problem, respectively. The student can interactively follow the steps of the algorithms in this module. We first outline the overall design of this collection. Then, we discuss the graph coloring module as well as the other available modules. We explain the new unpublished feature in which an animation of the algorithm is visualized. Finally, we explain the implementation details of this collection.

This dissertation is structured as follows. First, we discuss the known graph models from scientific computing in Chapter 2. Then, Chapter 3 discusses our new coloring heuristics. Chapter 4 introduces our interactive educational modules. Finally, the conclusion and future work are presented in Chapter 5.

# 2 Known graph models from scientific computing

In this chapter, we briefly discuss known graph formulations and models needed in this thesis. In each section of this chapter, we provide some references which explain further these concepts in details. We look at the ideas to determine the nonzeros of sparse Jacobian matrices in Section 2.1. We look at definitions of combining ILU preconditioning and partial Jacobian computation in Section 2.2. Throughout this thesis, we consider the natural ordering of the given matrix for ILU preconditioning.

## 2.1 Determining nonzeros of sparse Jacobian matrices

There are many references on exploiting the sparsity pattern of Jacobian matrices to improve the performance of automatic differentiation. Here, we look at full and partial Jacobian computation in Section 2.1.1 and Section 2.1.2.

### 2.1.1 Full Jacobian computation

Assume a program computes a function $f(x) : \mathbb{R}^n \to \mathbb{R}^m$ at the computational cost $t$. Techniques of automatic differentiation (AD) [1, 2] generate computer programs capable of evaluating the $m \times n$ Jacobian matrix $J$. The forward mode of automatic differentiation generates a program automatically which computes the product of the Jacobian matrix with a given seed matrix $V$, i.e., $JV$. There is a reverse mode of automatic differentiation which computes the product $WJ$ where $W$ is another seed matrix. These techniques of automatic differentiation compute the matrix-matrix products $JV$ and $WJ$ without assembling the Jacobian $J$.

Suppose the matrix $V$ has $c$ columns and the matrix $W$ has $r$ rows. The computational costs of these products using the forward and reverse modes is then given by $ct$ and $rt$, respectively. In general, the Jacobian $J$ is computed choosing either $c = n$ and $V$ as the identity of order $n$ in the forward mode or $r = m$ and $W$ as the identity of order $m$ in the reverse mode. However, if $J$ is sparse and its sparsity pattern is known, the number of columns of $V$ in the forward mode or the number of rows of $W$ in the reverse mode can be reduced to $c < n$ or $r < m$ such that all nonzero entries of $J$ still appear in the product $JV$ or $WJ$. This way, the computational cost is decreased using either the forward mode with an appropriate linear combination of the columns of $J$ or the reverse mode with a suitable linear combination of the rows of $J$; see the survey [11]. Later in this chapter,

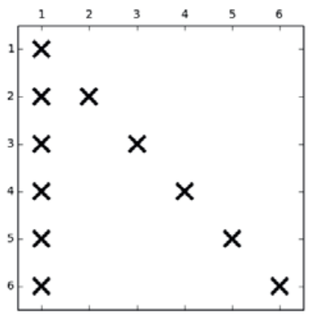
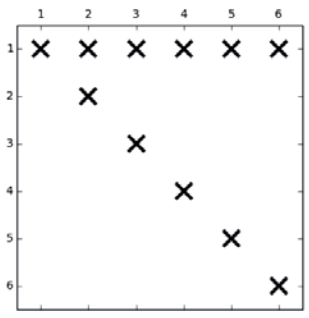
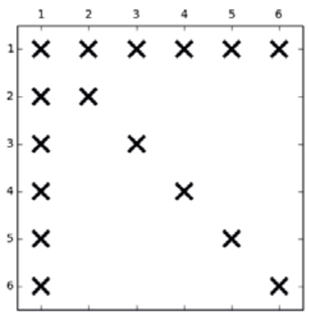




Figure 2.1: (Left) An example of a matrix compressed efficiently by columns. (Middle) An example of a matrix compressed efficiently by rows. (Right) An example of a matrix which cannot be compressed efficiently neither by columns nor by rows.

we formulate problems to compute the minimum values for $c$ and $r$ and the corresponding combinatorial problem.

### Scientific computing problem

Here, we find a seed matrix in which the corresponding number of rows and columns is smaller than the actual Jacobian matrix which is called compression. A unidirectional compression is a compression in either rows or columns in contrast to a bidirectional compression in which both rows and columns are compressed at the same time. The key idea behind this *unidirectional compression* is now illustrated for the forward mode. First, we present a definition as follows.

**Definition 1 (Structural Orthogonality)** *Let $J = [c_1, c_2, \dots, c_n]$ denote the $m \times n$ Jacobian matrix in which $c_i \in \mathbb{R}^m$ is the $i$th column. Two columns $c_i$ and $c_j$ are called* structurally orthogonal *if they do not have any nonzero element in a same row. Two columns are called* structurally non-orthogonal *if there is at least one row in which both columns, $c_i$ and $c_j$, have a nonzero element. Analogously, two rows are* structurally orthogonal *if they do not have any nonzero element in a same column.*

We can compute a linear combination of a group of structurally orthogonal columns of the Jacobian matrix such that this linear combination contains all elements of these columns. The definition of the structurally orthogonal columns can be similarly adapted to rows. It follows that the number of structurally orthogonal groups represents the computational cost either for columns or rows. Figure 2.1 (Left) and Figure 2.1 (Middle) show two examples of matrices which can be compressed efficiently by columns and by rows, respectively. Now, consider the matrix in Figure 2.1 (Right) that has neither structurally orthogonal columns nor structurally orthogonal rows. Therefore, there is no unidirectional compression of the matrix, neither by columns nor rows. However, the technique of bidirectional compression, which compresses both columns and rows at the same time, will reduce the computational cost for that example. This technique uses both forward and reverse modes of automatic differentiation.

For general sparsity patterns, it is not straightforward to figure out how to linearly combine columns and rows such that the computational cost is minimized. Hence, we introduce the combinatorial optimization problems 1 and 2 for unidirectional and bidirectional compression to determine the nonzero elements of large Jacobian matrices efficiently.

**Problem 1 (Minimum Unidirectional Compression)** *Let $J$ be a sparse $m \times n$ Jacobian matrix with a known sparsity pattern. Find a binary seed matrix $V$ of dimension $n \times c$ whose number of columns is minimized such that all nonzero elements of $J$ also appear in the matrix-matrix product $JV$.*

The corresponding compression problem for minimizing the number of rows in the matrix-matrix product $WJ$ is straightforward and omitted here.

**Problem 2 (Minimum Bidirectional Compression)** *Let $J$ be a sparse $m \times n$ Jacobian matrix with known sparsity pattern. Find a pair of binary seed matrices $V$ of dimension $n \times c$ and $W$ of dimension $r \times m$ in which the number of columns of $V$ and the number of rows of $W$ sum up to a minimal value, $c + r$, such that all nonzero elements of $J$ also appear in the pair of matrix-matrix products $JV$ and $WJ$.*

### Combinatorial model

We reformulate the scientific computing problems which we have discussed in Section 2.1.1. The new formulation is an equivalent problem defined on a carefully chosen graph model. The survey [11] discusses different methods to exploit the sparsity involved in derivative computations. We first look at a simple graph model for the unidirectional compression.

**Definition 2 (Column Intersection Graph)** *The column intersection graph $G = (V, E)$ associated with an $n \times n$ Jacobian matrix $J$ consists of a set of vertices $V = \{v_1, v_2, \ldots, v_n\}$ whose vertex $v_i$ represents the $i$th column $J(:, i)$. Furthermore, there is an edge $(v_i, v_j)$ in the set of edges $E$ if and only if the columns $J(:, i)$ and $J(:, j)$ represented by $v_i$ and $v_j$ are structurally non-orthogonal.*

As we have a graph model associated with our Jacobian matrix in Definition 2, the grouping of columns can be encoded in the following well-known graph coloring problem.

**Definition 3 (Coloring)** *A coloring of $G = (V, E)$ is a mapping $\Phi : V \to \{1, \ldots, p\}$ with the property $\Phi(v_i) \neq \Phi(v_j)$ if $(v_i, v_j) \in E$.*

Coleman and Moré [4] then showed that Problem 1, which asks for a seed matrix with a minimal number of columns, is equivalent to the following coloring problem.

**Problem 3 (Minimum Coloring)** *Find a coloring $\Phi$ of the column intersection graph $G$ associated with a sparse Jacobian $J$ with a minimal number of colors.*

Although, this model is convincing for the unidirectional compression, the bidirectional compression can not be an instance of this model. A bidirectional compression needs the information of both rows and columns. Therefore, a bipartite graph model is defined for this purpose as in [6, 21, 7].

**Definition 4 (Bipartite Graph Model)** *In the bipartite graph model, the vertex set $V = V_c \cup V_r$ is decomposed into a set of vertices $V_c$ representing columns of $J$ and another set of vertices $V_r$ representing rows. The set of edges $E$ is used to represent the nonzero elements and it is defined as follows. An edge $(c_i, r_j) \in E$ connects a column vertex $c_i \in V_c$ and a row vertex $r_j \in V_r$ if there is a nonzero element in $J$ at the position represented by $c_i$ and $r_j$. The graph is bipartite indicating that all edges connect vertices from one set $V_c$ to the other set $V_r$. That is, there is no edge connecting vertices within the set $V_c$ or within $V_r$. Moreover, two vertices that are connected by a path of length two, are called* distance-2 neighbors*.*

The coloring problem in the column intersection graph can also be represented in this bipartite graph model. This equivalent coloring is done only in the set of column vertices. Also, *distance*-2 *neighbors* should be considered instead of adjacent vertices.

The overall idea behind transforming Problem 2, Minimum Bidirectional Compression, into an equivalent problem using the bipartite graph model is as follows. The grouping of the columns and rows is expressed by representing each group by a color. Vertices that belong to the same group of columns/rows are assigned the same color. Formally, this is represented by a coloring of a bipartite graph. Such a coloring is a mapping

$$\Phi : V_c \cup V_r \rightarrow \{0, 1, \ldots, p\}$$

that assigns to each vertex a color represented by an integer. The coloring $\Phi$ also involves a "neutral" color representing the following "don't color" situation. A vertex $v \in V_c \cup V_r$ that is not used in the grouping of columns/rows is assigned the neutral color $\Phi(v) = 0$. More precisely, if $\Phi(v) = 0$ for a column vertex $v$ then every nonzero represented by an incident edge of $v$ is determined by a linear combination of rows. Similarly, a nonzero entry represented by an edge that is incident to a neutrally-colored row vertex is determined by a linear combination of columns.

To represent the process of finding seed matrices using the bipartite graph model, it is necessary to consider the underlying properties, which are as follows:

1. The computational cost roughly consists of the number of groups of structurally orthogonal columns and rows. Since the overall cost is the sum of the costs associated with the forward mode and the reverse mode, the (non-neutral) colors for the forward mode and the (non-neutral) colors for the reverse mode need to be different.

2. It may happen that some nonzero elements may be computed twice, by the forward mode in $JV$ and by the reverse mode in $WJ$. Therefore, an edge representing such a nonzero element connects two vertices with two different non-neutral colors. In

general, since the Minimum Bidirectional Compression problem asks for computing *all* nonzero elements, at least one vertex of every edge has to be colored with a non-neutral color.

3. Suppose two columns are structurally non-orthogonal and have a nonzero element in a same row. If this row is not handled by the reverse mode, these two columns need to be in different column groups. The same argument holds for corresponding situations with row groups.

4. Consider three nonzero elements in the matrix positions $(i,k)$, $(i,\ell)$, and $(j,k)$. Suppose that the nonzero at $(i,k)$ is computed by the reverse mode assigning some (non-neutral) color to the row vertex $r_i$. Then, if $(j,k)$ is also computed via the reverse mode, a second (non-neutral) color is needed for $r_j$. Now, if $(i,\ell)$ is already determined by the reverse mode for the row $i$ the column vertex $c_\ell$ is assigned the neutral color. However, if $(i,\ell)$ is computed by the forward mode, a third (non-neutral) color is needed for $c_\ell$. A similar argument holds if $(i,k)$ is computed by the forward mode.

Based on these considerations, the following definition captures these properties.

**Definition 5 (Star Bicoloring)** *Given a bipartite graph $G = (V_c \cup V_r, E)$, then a mapping $\Phi : V_c \cup V_r \to \{0, 1, \ldots, p\}$ is a star bicoloring of $G$ if the following conditions are satisfied:*

1. *Vertices in $V_c$ and $V_r$ receive disjoint colors, except for the neutral color $0$. That is, for every $c_i \in V_c$ and $r_j \in V_r$, either $\Phi(c_i) \neq \Phi(r_j)$ or $\Phi(c_i) = \Phi(r_j) = 0$.*

2. *At least one vertex of every edge receives a non-neutral color. That is, for every $(c_i, r_j) \in E$, the conditions $\Phi(c_i) \neq 0$ or $\Phi(r_j) \neq 0$ hold.*

3. *For every path $(u, v, w)$ with $\Phi(v) = 0$, the condition $\Phi(u) \neq \Phi(w)$ is satisfied.*

4. *Every path of length three with four vertices uses at least three colors (possibly including the neutral color).*

Using the bipartite graph model and the definition of a star bicoloring, the problem Minimum Bidirectional Compression is equivalent to the following graph problem.

**Problem 4 (Minimum Star Bicoloring)** *Given the bipartite graph $G = (V_r \cup V_c, E)$ associated with a sparse Jacobian matrix $J$, find a star bicoloring of $G$ with a minimal number of non-neutral colors.*

A unidirectional compression is a special case of a bidirectional compression. More precisely, a unidirectional compression with respect to columns corresponds to a star bicoloring in which all the vertices in $V_c$ are colored with a non-neutral color and all row vertices are colored with the neutral color. This way, the coloring constraint of a star bicoloring reduces to the coloring of distance-2 neighbors in the bipartite graph using different (non-neutral) colors. This distance-2 coloring in the bipartite graph model is then equivalent to a coloring in the undirected graph model in which all neighbors are colored differently. Finally, a discussion of the computational complexity of Problem 4 including recent new results is given in [22].

### 2.1.2 Partial Jacobian computation

Gebremedhin et al. [11] introduced the concept of partial Jacobian computation in which only a subset of the nonzero Jacobian entries, the required elements, are to be determined. Lülfesmann [14] studied this area in more details and introduced some heuristics for partial computation.

#### Scientific computing problem

Let $R$ be the set representing required elements. The definition of *structural orthogonality* is adapted for partial Jacobian computation as follows.

**Definition 6 (Partially Structural Orthogonality)** *Two columns $c_i$ and $c_j$ are partially structurally orthogonal with respect to $R$ if and only if they do not have a nonzero element in a same row where at least one of these nonzero elements is required.*

The corresponding combinatorial optimization problem for the unidirectional and bidirectional compression restricted to the required elements can be formulated as follows.

**Problem 5 (Minimum Partial Unidirectional Compression)** *Let $J$ represents a sparse $m \times n$ Jacobian matrix with known sparsity pattern and $R$ be a subset of the nonzero elements of $J$. Find a binary seed matrix $V$ of dimension $n \times c$ whose number of columns is minimized such that all nonzero elements of $R$ also appear in the matrix-matrix product $JV$.*

**Problem 6 (Minimum Partial Bidirectional Compression)** *Let $J$ represents a sparse $m \times n$ Jacobian matrix with known sparsity pattern and $R$ be a subset of the nonzero elements of $J$. Find a pair of binary seed matrices $V$ of dimension $n \times c$ and $W$ of dimension $r \times m$ in which the number of columns of $V$ and the number of rows of $W$ sum up to a minimal value, $c + r$, such that all nonzero elements of $R$ also appear in the pair of matrix-matrix products $JV$ and $WJ$.*

Now, we discuss an equivalent graph-theoretical formulation of this problem.

### Combinatorial model

Based on [11, 14], the definitions of full Jacobian coloring are adapted for the restricted colorings as follows.

**Definition 7 (Restricted distance-2 coloring)** *Given a bipartite graph $G = (V_c \cup V_r, E)$ and a subset of required edges $E_R \subseteq E$, then a mapping $\Phi : V_c \to \{0, 1, \dots, p\}$ is a distance-2 coloring of G restricted to $E_R$ if all column vertices incident to at least one required edge $e \in E_R$ get a nonzero color and for every path $(c_k, r_i, c_j)$ with $c_k, c_j \in V_c$, $r_i \in V_r$, and $(r_i, c_j) \in E_R$, $\Phi(c_k) \neq \Phi(c_j)$.*

**Definition 8 (Restricted star bicoloring)** *Given a bipartite graph $G = (V_c \cup V_r, E)$ and a subset of required edges $E_R \subseteq E$, then a mapping $\Phi : V_c \cup V_r \to \{0, 1, \dots, p\}$ is a star bicoloring of G restricted to $E_R$ if the following conditions are satisfied:*

1. *Vertices in $V_c$ and $V_r$ receive disjoint colors, except for the neutral color 0. That is, for every $c_i \in V_c$ and $r_j \in V_r$, either $\Phi(c_i) \neq \Phi(r_j)$ or $\Phi(c_i) = \Phi(r_j) = 0$.*
2. *At least one end point of an edge in $E_R$ receives a nonzero color.*
3. *For every edge $(r_i, c_j) \in E_R$, $r_i, r_l \in V_r$, and $c_j, c_k \in V_c$,*
   - *if $\Phi(r_i) = 0$, then for every path $(c_k, r_i, c_j)$, $\Phi(c_k) \neq \Phi(c_j)$*
   - *if $\Phi(c_j) = 0$, then for every path $(r_i, c_j, r_l)$, $\Phi(r_i) \neq \Phi(r_l)$*
   - *if $\Phi(r_i) \neq 0$ and $\Phi(c_j) \neq 0$, then for every path $(c_k, r_i, c_j, r_l)$, $\Phi(c_k) \neq \Phi(c_j)$ or $\Phi(r_i) \neq \Phi(r_l)$*

Now, the optimization problems are formulated as follows.

**Problem 7 (Minimum Restricted Distance-2 Coloring)** *Given the bipartite graph $G = (V_r \cup V_c, E)$ associated with a sparse Jacobian matrix $J$ and a set of required edges $E_R$, find a distance-2 coloring of G restricted to $E_R$ with a minimal number of non-neutral colors.*

**Problem 8 (Minimum Restricted Star Bicoloring)** *Given the bipartite graph $G = (V_r \cup V_c, E)$ associated with a sparse Jacobian matrix $J$ and a set of required edges $E_R$, find a star bicoloring of G restricted to $E_R$ with a minimal number of non-neutral colors.*

## 2.2 Combining partial Jacobian computation and ILU

Given a large sparse nonsingular $n \times n$ Jacobian matrix $J$, we are considering the solution to the following system of linear equations,

$$Jx = b,$$

in which $x$ and $b$ are $n \times 1$ vectors. Iterative solvers are considered to be among the effective solution techniques [23].

Iterative techniques are typically used in combination with the preconditioning techniques [24, 23]. Rather than solving the previous system, we can solve the preconditioned system

$$M^{-1}Jx = M^{-1}b, \tag{2.1}$$

where the $n \times n$ matrix $M$ serves as a preconditioner that approximates the coefficient matrix,

$$M \approx J.$$

Some preconditioning techniques like ILU preconditioning can generate a preconditioner which has nonzero at some places in which the Jacobian matrix $J$ has zero elements. These nonzero elements are called fill-in.

### 2.2.1 Scientific computing problem

Today, there is no general and established strategy on how to combine automatic differentiation with preconditioning. The reason is that standard preconditioning techniques typically need access to individual nonzero elements of the coefficient matrix whereas automatic differentiation gives efficient access to a different level of granularity, namely rows or columns.

Common approaches to constructing the preconditioner $M$ are based on accessing individual nonzero entries $J(i,j)$ of the Jacobian. For instance, diagonal scaling consists of the diagonal matrix $M$ whose diagonal entries $M(i,i)$ are equal to $J(i,i)$ for all $i = 1, 2, \ldots, n$. Another option is to compute a decomposition of the form

$$M = LU$$

where $L$ is a unit lower triangular matrix and $U$ is an upper triangular matrix resulting from performing Gaussian elimination on $J$ and dropping out nonzero elements that would be generated by this process in certain predetermined positions. Similar to diagonal scaling, this incomplete LU factorization (ILU) needs access to individual nonzero entries of $J$ or segments of rows/columns of $J$. In general, accessing an individual nonzero entry via automatic differentiation is as efficient as accessing a complete column or row. In practice, an access to some individual nonzero entry is therefore prohibitively expensive regarding computing time.

Lülfesmann [14] introduced a new approach in which the required elements of the Jacobian matrix $J$ are given in the form of the nonzero elements within $k \times k$ blocks on the main diagonal. Then, the preconditioner is built based on these selected nonzero required elements instead of the whole Jacobian matrix. Building a preconditioner based on a subset of the nonzero elements is explained in [25]. We call these required elements *initially* required elements represented by $R_i$ since they are an initial set for the further process. Now, the coloring problem can be solved restricted to the set of initially required nonzero elements $R_i$.

The result of a coloring algorithm groups columns and rows together. In this process, the required elements of $J$ are always computed. The remaining nonzero elements of $J$, which are called the nonrequired elements, are divided into two sets of elements: the elements which are computed and the ones which would be eliminated. We can think of these computed nonrequired nonzero elements as byproducts of the coloring and computing the matrix-matrix products $JV$ and $WJ$. Since the number of colors does not change, the idea is to add these extra byproducts also to $R_i$. So, we call these byproduct elements the *potentially* required elements $R_p$ which are specified by

$$R_p \subset \mathrm{pat}(J) - R_i \quad \text{so that} \quad |\Phi(R_i)| = |\Phi(R_i \cup R_p)|,$$

where $\mathrm{pat}(J)$ represents the nonzero pattern of the Jacobian matrix $J$ and $|\Phi(R_i)|$ is the number of colors resulting from the coloring restricted to $R_i$.

As we have discussed, an ILU preconditioner applied to $R_i$ can produce fill-in. A subset $R_a$ of potentially required elements, which is called the set of *additionally* required elements, is selected such that no new fill-in is generated. These elements can be added to the initially required elements for further computation. The additionally required elements are formulated as follows.

$$R_a \subset R_p \quad \text{so that} \quad SILU(R_i) \cup R_a = SILU(R_i \cup R_a),$$

in which $SILU$ means the symbolic ILU factorization.

Now, we formulate two optimization problems in this thesis as follows.

**Problem 9 (Maximum Potentially Required Elements)** *Let $J$ be a sparse $m \times n$ Jacobian matrix with known sparsity pattern and $R_i$ is a set of required elements. Find a set of potentially required elements $R_p$ with maximal cardinality.*

**Problem 10 (Maximum Additionally Required Elements)** *Let $J$ be a sparse $m \times n$ Jacobian matrix with known sparsity pattern and $R_i$ is a set of required elements. Find a set of additionally required elements $R_a$ with maximal cardinality.*

The overall approach applied to a specific problem from aerodynamics is detailed in [15].

### 2.2.2 Combinatorial model

The bipartite graph model presented in Definition 4 is a suitable model to find new algorithms for the initially, potentially, and additionally required elements. Recall that each edge in the bipartite graph model is related to a nonzero element in the matrix. Then, given a bipartite graph $G = (V, E)$, three subsets $E_i, E_p, E_a \subset E$ are considered for the initially, potentially, and additionally required elements, respectively. Given the $k \times k$ block on the diagonal as required elements $R_i$ and the Jacobian matrix $J$, here is a list of steps in the computation of additionally required elements and the corresponding algorithm on the bipartite graph model. More details can be found in [14].

```
function pot_d2_coloring($G = (V_r \cup V_c, E)$, $E_i \subseteq E$, $\phi$)
  $E_p = \emptyset$
  for $(r_i, c_j) \in E - E_i$ with $\Phi(c_j) \neq 0$
    for $c_k \in N_1(r_i, G)$ with $j \neq k$ and $(r_i, c_k) \notin E_i$
      if $\Phi(c_j) = \Phi(c_k)$
        continue with next edge $(r_i, c_j) \in E - E_i$
    $E_p = E_p \cup \{(r_i, c_j)\}$
  return $E_p$
```

Algorithm 2.1: Find potentially required elements for distance-2 coloring (based on the algorithm 4.2 from [14]).

- choose the initially required elements $R_i$.
- setup the bipartite graph $G$ from the Jacobian matrix $J$ and the subset $E_i \subset E$ representing $R_i$.
- compute the coloring $\Phi(E_i)$ of the bipartite graph $G$ restricted to $E_i$.
- find the set of potentially required elements $E_p \subset E - E_i$ such that $|\Phi(E_i)| = |\Phi(E_i \cup E_p)|$.
- find the set of additionally required elements $E_a \subset E_p$ such that $SILU(R_i) \cup R_a = SILU(R_i \cup R_a)$.

Algorithm 2.1 and Algorithm 2.2 show the algorithms to compute the potentially required elements for the distance-2 coloring and star bicoloring, respectively. In these algorithms, $N_1$ means the distance-1 neighbors (the adjacent vertices). Given a bipartite graph $G$, the required edges $E_i$, and a distance-2 coloring $\Phi$, Algorithm 2.1 computes a set of potentially required elements. This algorithm iterates over nonrequired edges ($E - E_i$) and checks if such an edge can be added to the set of potentially required elements. Similarly, Algorithm 2.2 computes the potentially required elements when the given coloring $\phi$ is a star bicoloring.

To compute the additionally required elements, a formulation of ILU preconditioning in the language of graphs is needed. Hysom and Pothen [26] introduced a graph model for the incomplete LU factorization in which the matrix is the adjacency matrix of this graph. It should be considered that the graph would be directed if the matrix is not symmetric. The concept of *fill path* is defined to characterize the fill-in in ILU preconditioning,

**Definition 9 (Fill path)** *A fill path is a path $(v_i, ..., v_k, ..., v_j)$ with $k < \min(i, j)$. It means the index of all the inner nodes in a given ordering of vertices is smaller than the indices of the vertices $v_i$ and $v_j$.*

It follows that a matrix element $(i, j)$ is a fill-in if and only if there is a fill path between $v_i$ and $v_j$. In addition to the concept of fill path, another concept of fill level $l$ is needed to formulate the level-based incomplete LU factorization [26]. This parameter is used to

```
function pot_star_bicoloring($G = (V_r \cup V_c, E)$, $E_i \subseteq E$, $\phi$)
  $E_p = \emptyset$
  for $(r_i, c_j) \in E - E_i$ with $\Phi(r_i) \neq 0$ or $\Phi(c_j) \neq 0$
    if $\Phi(r_i) = 0$
      for $c_k \in N_1(r_i, G)$ with $j \neq k$ and $(r_i, c_k) \notin E_i$
        if $\Phi(c_j) = \Phi(c_k)$
          continue with the next edge $(r_i, c_j) \in E - E_i$

    if $\Phi(c_j) = 0$
      for $r_l \in N_1(c_j, G)$ with $j \neq l$ and $(r_l, c_j) \notin E_i$
        if $\Phi(r_i) = \Phi(r_l)$
          continue with the next edge $(r_i, c_j) \in E - E_i$

    if $\Phi(r_i) \neq 0$ and $\Phi(c_j) \neq 0$
      for $c_k \in N_1(r_i, G)$ with $i \neq k$
        for $r_l \in N_1(c_j, G)$ with $j \neq l$
          if $\Phi(c_j) = \Phi(c_k)$ and $\Phi(r_i) = \Phi(r_l)$
            continue with the next edge $(r_i, c_j) \in E - E_i$

    $E_p = E_p \cup \{(r_i, c_j)\}$
  return $E_p$
```

Algorithm 2.2: Find potentially required elements for star bicoloring (based on Algorithm 4.3 from [14]).

filter the generated fill-in. This parameter is the length of the fill path. In the level-based incomplete LU factorization, the generated fill-in is allowed to be considered only up to the level $l$. We fix this level parameter to 2 throughout this thesis.

Lülfesmann [14] adapted a corresponding bipartite graph model for ILU preconditioning. The concept of fill path is also redefined for the bipartite graph model in which a path is replaced by a distance-2 path.

**Definition 10 (Fill path in bipartite graph)** *A path $(r_i, c_k, r_k, c_l, r_l, ..., c_j)$ in the bipartite graph model starting from a row vertex is a fill path if and only if all vertices between $r_i$ and $c_j$ have a lower index than $i$ and $j$.*

Again, it follows that there is a fill-in $(i, j)$ in the matrix if and only if there is a fill path between $r_i$ and $c_j$ in the corresponding bipartite graph.

Based on the bipartite graph models for coloring and the ILU preconditioning, Lülfesmann [14] proposed two algorithms to compute the additionally required elements: conservative and sophisticated. We consider the sophisticated algorithm shown in Algorithm 2.3 to compute the additionally required elements throughout this thesis. This algorithm looks at each potentially required edge. If it is possible that a fill path is generated by adding this edge, the edge is not considered for the set of additionally required edges. In this sophisticated approach, the algorithm iterates once again over all potentially required elements. This process is repeated until no new additionally required edges are found.

```
function add($G = (V_r \cup V_c, E)$, $E_i \subseteq E$, $E_p$, $E_F$)
  $E_a = \emptyset$
  do
    for $(r_i, c_j) \in E_p$
      if $i > j$
        for $c_l \in N_1(r_j, G[E_i \cup (E_F \cup E_a)])$ with $l > j$
          if $(r_i, c_l) \notin E_i \cup (E_F \cup E_a)$
            continue with next edge $(r_i, c_j) \in E_p$
      else if $i \leq j$
        for $r_k \in N_1(c_i, G[E_i \cup (E_F \cup E_a)])$ with $k > i$
          if $(r_k, c_j) \notin E_i \cup (E_F \cup E_a)$
            continue with next edge $(r_i, c_j) \in E_p$
      $E_a = E_a \cup \{(r_i, c_j)\}$
      $E_p = E_p - \{(r_i, c_j)\}$
  while $|E_a|$ is increased in the last iteration
  return $E_a$
```

Algorithm 2.3: Find additionally required elements (based on Algorithm 4.5 from [14]).

# 3 New coloring heuristics

Karp [27] proves that the graph coloring problem is NP-complete for general graphs. Hence, various coloring heuristics are studied throughout the years with a polynomial complexity. Greedy coloring is a widely used heuristic which has a low computational complexity and computes a *reasonable* coloring (see [28]). The greedy coloring algorithm for the restricted distance-2 coloring is given in Algorithm 3.1 which is adapted from Lülfesmann [14]. The computed coloring $\Phi : V_c \to \{1, 2, ..., p\}$ on the vertex set $V_c = \{1, 2, ..., n\}$ is restricted to the initially required edges $E_i$. All arrays in all algorithms, like $forbiddenColors$, implemented in this thesis start with the index 0. The assignment $forbiddenColors[c] = v$ means the vertex $v$ can not be colored by the color $c$. Since this algorithm does not consider the color 0, the element $forbiddenColors[0]$ is ignored in line 7 of Algorithm 3.1.

In this algorithm, the function $N_2(v, E_i)$ finds all distance-2 neighbors of $v$ restricted to the set of initially required edges $E_i$. More clearly, the function $N_2(v, E_i)$ computes all vertices on all paths of length 2 from the vertex $v$ such that at least one edge of such a path is in the set $E_i$,

$$N_2(v, E_i) = \{z \in V : \text{ there is a path } (v, w, z) \text{ with } (v, w) \in E_i \text{ or } (w, z) \in E_i \text{ or both.}\}$$

To color a vertex $v$ we iterate over all its distance-2 neighbors in $N_2(v, E_i)$ which are already colored. Remember that these assigned colors can not be used to color the vertex $v$. After collecting all these forbidden colors, we assign to $v$ the color with the smallest index different from these forbidden colors. It follows that the computational complexity of this algorithm is $\mathcal{O}(n\Delta^2)$ in which $n$ and $\Delta$ are the number of vertices and the maximum vertex degree, respectively. Recall from the last chapter that the Jacobian matrices are sparse. Therefore, $n$ is large and $\Delta$ and $\Delta^2$ are relatively small.

```
1  function d2_color($G = (V_r \cup V_c, E)$, $E_i \subseteq E$)
2    $\Phi \leftarrow [0 \ldots 0]$
3    $forbiddenColors \leftarrow [0 \ldots 0]$
4    for $v \in V_c$ with $\exists r \in V_r : (v, r) \in E_i$
5      for $n \in N_2(v, E_i)$ with $\Phi(n) \neq 0$
6          $forbiddenColors[\Phi(n)] = v$
7      $\Phi(v) = \min\{a > 0 : forbiddenColors[a] \neq v\}$
8    return $\Phi$
```

Algorithm 3.1: The greedy algorithm for the distance-2 coloring restricted to the edge set $E_i$ for columns.

| Matrix | Size | Nonzeros | Symmetric |
|---|---|---|---|
| *steam1.mtx* | $240 \times 240$ | 2248 | false |
| *steam2.mtx* | $600 \times 600$ | 5660 | false |
| *685_bus.mtx* | $685 \times 685$ | 3249 | true |
| *nos3.mtx* | $960 \times 960$ | 15844 | true |
| *ex7.mtx* | $1633 \times 1633$ | 46626 | false |
| *ex33.mtx* | $1733 \times 1733$ | 22189 | true |
| *orani678.mtx* | $2529 \times 2529$ | 90158 | false |
| *cavity16.mtx* | $4562 \times 4562$ | 137887 | false |
| *crystm01.mtx* | $4875 \times 4875$ | 105339 | true |
| *rajat01.mtx* | $6833 \times 6833$ | 43250 | false |
| *gyro_m.mtx* | $17361 \times 17361$ | 340431 | true |
| *ford2.mtx* | $100196 \times 100196$ | 544688 | true |
| *cage3.mtx* | $5 \times 5$ | 19 | false |
| *cage4.mtx* | $9 \times 9$ | 49 | false |
| *cage5.mtx* | $37 \times 37$ | 233 | false |
| *cage6.mtx* | $93 \times 93$ | 785 | false |
| *cage7.mtx* | $340 \times 340$ | 3084 | false |
| *cage8.mtx* | $1015 \times 1015$ | 11003 | false |
| *cage9.mtx* | $3534 \times 3534$ | 41594 | false |
| *cage10.mtx* | $11397 \times 11397$ | 150645 | false |
| *cage12.mtx* | $130228 \times 130228$ | 2032536 | false |

Table 3.1: Throughout this chapter, the numerical experiments are carried out using these matrices from the Florida sparse matrix collection.

In this algorithm, vertices are colored one at a time. Therefore, the vertex ordering plays a major role in the greedy algorithm. Hence, there are many publications on how to choose a suitable ordering for a serial or parallel version of coloring [29, 30]. Various orderings are studied for coloring heuristics throughout the years. Here are some orderings for coloring which are applied before doing the coloring: the largest-first ordering (LFO) [31], the incidence-degree ordering (IDO) [32], the saturation-degree ordering (SDO) [30], and the smallest-last ordering (SLO) [29].

This chapter is structured as follows. We modify the greedy algorithm in Section 3.1 such that the number of potentially and additionally required elements are increased. Later, we discuss a better heuristic for the special case of coloring restricted to a diagonal in Section 3.3. Finally, we introduce our computational package *PreCol* in Section 3.4 that computes the unidirectional and bidirectional restricted colorings with different algorithms as well as the ILU preconditioning.

Throughout this chapter, the numerical experiments are carried out using the matrices in Table 3.1 from the Florida sparse matrix collection [33].

## 3.1 Maximizing the set of additionally required elements

Here, our focus is to solve the optimization Problems 9 and 10. As we have discussed, there are nonrequired nonzero elements which are also computed as a by-product of the computation of required elements. The nonrequired elements have a major effect on the determination of potentially required elements and additionally required elements. The following example illustrates that a modified coloring can increase the number of nonrequired elements which are determined.

$$\begin{pmatrix} * & * & * \\ 0 & r & r \\ * & 0 & * \end{pmatrix} \qquad \begin{pmatrix} * & * & * \\ 0 & r & r \\ * & 0 & * \end{pmatrix} \tag{3.1}$$

Here, the symbol $r$ stands for a required element, the symbol $*$ stands for other nonzero elements (the nonrequired elements), and the number 0 denotes a zero element.

If the first and second columns get the same color as given in the left matrix, we will get the nonzero at position $(3, 1)$ as a by-product. However, there are certain degrees of freedom. In this example, one could also assign the same color to columns 1 and 3 as illustrated in the right matrix in which no nonzero element in the last row will be computed as a by-product. This idea leads to the problem of maximizing the number of nonrequired nonzero elements that are computed as a by-product. Here, we introduce a new heuristic which increases the number of determined nonrequired elements while the number of colors remains almost the same.

### 3.1.1 Restricted distance-2 coloring

Our goal here is to increase the number of potentially and additionally required elements. Given a bipartite graph $G = (V_c \cup V_r, E)$ and a vertex $v \in V_c$, we define a function $L_v : S \subseteq V_c \to \mathbb{N}$. The function $L_v(w)$ computes the number of nonrequired elements that are actually determined by the linear combination of the columns corresponding to the vertices $v$ and $w \in V_c$. We call these elements the determined nonrequired elements. Here, we focus on the unidirectional compression for columns represented by $V_c$. However, an analogous discussion holds for the unidirectional compression for rows.

We modify the greedy algorithm presented in Algorithm 3.1 as follows. For this new heuristic, we define two operators. Given a function $f : A \to B$ and a subset $S \subseteq A$ the operators arg max and arg min are defined as,

$$\arg\max_{x \in S} f(x) = \{x \mid \forall y \in S : f(y) \leq f(x)\},$$
$$\arg\min_{x \in S} f(x) = \{x \mid \forall y \in S : f(y) \geq f(x)\}.$$

Algorithm 3.2 shows this new heuristic. In this algorithm, we iterate over all uncolored vertices. First, we color the vertex $v$ with a color different form its distance-2 neighbors restricted to $E_i$ like the previous greedy coloring. Note, however, that there is an additional

```
1   function d2_color_nreq($G = (V_r \cup V_c, E), E_i \subseteq E$)
2     $\Phi \leftarrow [0 \dots 0]$
3     $forbiddenColors \leftarrow [0 \dots 0]$
4     for $v \in V_c$ with $\exists r \in V_r : (v, r) \in E_i$ and $\Phi(v) = 0$
5       for $n \in N_2(v, E_i)$ with $\Phi(n) \neq 0$
6         $forbiddenColors[\Phi(n)] = v$
7       $\Phi(v) = \min\{a > 0 : forbiddenColors[a] \neq v\}$
8
9       $I_v = \{z \in V_c : z \neq v$ and $z \notin N_2(v)$ and $\Phi(z) = 0\}$
10      if $I_v \neq \emptyset$
11        $maxs = \arg\max_{x \in I_v} L_v(x)$
12        $\Phi(maxs[0]) = \Phi(v)$
13    return $\Phi$
```

Algorithm 3.2: New coloring heuristic for distance-2 coloring considering the nonrequired elements.

$$\begin{pmatrix} r & 0 & 0 & 0 \\ 0 & 0 & r & 0 \\ 0 & * & * & 0 \\ 0 & r & 0 & r \end{pmatrix}$$

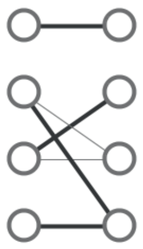


Figure 3.1: An example of a Jacobian and its bipartite graph model in which Algorithm 3.2 does not generate a valid coloring.

condition which ensures that the vertex $v$ has not been previously colored. After the coloring of $v$, Algorithm 3.2 finds the set $I_v$ representing the uncolored vertices that can have the same color as the vertex $v$. Additionally, we compute $L_v(x)$ for each vertex $x \in I_v$ and color a vertex $w \neq x$ with the same color as $v$ if $w$ leads to the maximum value of $L_v(w)$. The time complexity of this new heuristic is estimated as follows. The outer loop at most consists of $n$ vertices. For a vertex $v$, the complexity of coloring in line 5 to 7 is $\mathcal{O}(\Delta^2)$ as in Algorithm 3.1. The set $I_v$ is computed in $\mathcal{O}(n)$. Since the computation of $L_v(w)$ is $\mathcal{O}(n)$, the computation of the set $maxs$ is $\mathcal{O}(n^2)$. Thus, the general complexity is given by $\mathcal{O}(n^3)$.

Although we color a vertex $v$ and another vertex $u \in I_v$ in each iteration it does not always mean that the coloring is valid. Consider the Jacobian and its bipartite graph model in Figure 3.1 where the required edges are the bold edges in the graph.

Here, we assume that the outer loop is executed in a natural ordering corresponding to iterate over the columns of the Jacobian from left to right. After the first vertex $v_1$ corresponding to the first column is given some color, the set $I_v$ consists of the columns 2, 3, and 4. Since all these three columns can be compressed with the first column, the function $L_{v_1}(x)$ is given by the number of nonrequired elements in a column corresponding

```
function is_valid($G = (V_r \cup V_c, E)$, $E_i \subseteq E$)
    for $v \in V_c$ and $n \in N_2(v, E_i)$
      if $\Phi(n) = \Phi(v)$
        return false
  return true
```

Algorithm 3.3: A function which checks the validation of the coloring.

to $x$. Thus, the values of $L_{v_1}(x)$ for the columns 2, 3, and 4 are 1, 1, 0, respectively. Therefore, the first and the second column are colored identically. In the next iteration of the outer loop, the third column is then colored with the same color as the first column since the vertex $v_3$ corresponding to the third column does not have any distance-2 neighbors restricted to $E_i$. Thus, the array $forbiddenColors$ is not changed in this iteration. So, the colors of first three columns are the same. Now, the set $I_{v_3}$ contains the vertex $v_4$ corresponding to the fourth column since the others are already colored. Therefore, the fourth column gets the color of the third column. However, this is not a valid coloring since the fourth and second columns can not be compressed. Considering this problem, we check if the coloring is valid at the end of the algorithm by the function given in Algorithm 3.3. An observation is that the coloring is valid for all the matrices that are mentioned in this thesis.

Table 3.2 presents the number of potentially and additionally required elements computed by Algorithm 3.1 and Algorithm 3.2 and for different orderings for coloring. Table 3.2 (Top), (Middle), and (Bottom) are the results for the natural ordering, the LFO ordering, and the SLO ordering, respectively. The size of the diagonal blocks is fixed to 10. In these tables, the numbers of both potentially and additionally required elements tend to increase in the new proposed Algorithm 3.2 regardless of the ordering. An observation is that, for the matrix *pesa.mtx* with the ordering LFO, the number of potentially required elements decreases while the number of additionally required elements increases. Also, both the numbers of potentially and additionally required elements decrease in Algorithm 3.2 for *ex7.mtx* and for all orderings.

Now, we compare the results by changing the block size varying from 1 to 70. We compute Algorithm 3.1 and Algorithm 3.2 for the matrices *ex33* and *crystm01* with the three different orderings: the natural ordering, the LFO ordering, and the SLO ordering. Additionally, we compute the number of colors in each case too. All figures are illustrated in Appendix A.1. An observation is that the behavior of the figures of the potentially required elements and the additionally required elements is similar. Hence, we consider only the additionally required elements for $ex33$ now. The number of additionally required elements computed by Algorithm 3.2 with the ordering SLO is overall larger than Algorithm 3.1 (see Figure 3.2). The number of colors remains almost the same for the both colorings (see Figure 3.3). Here, Algorithm 3.2 leads to a larger number of colors compared to Algorithm 3.1. However, there are also orderings where Algorithm 3.2 is not superior for all block sizes in terms of the number of additionally required elements. For example, Figure 3.4 shows the number of additionally required elements for the natural ordering. Here,

| Matrix (NAT) | $\|R_p\|$ | | $\|R_a\|$ | |
|---|---|---|---|---|
| | Algorithm 3.1 | Algorithm 3.2 | Algorithm 3.1 | Algorithm 3.2 |
| *steam1.mtx* | 64 | 786 | 64 | 630 |
| *steam2.mtx* | 240 | 1880 | 240 | 1400 |
| *nos3.mtx* | 1638 | 6756 | 1106 | 4296 |
| *crystm01.mtx* | 17822 | 47556 | 10388 | 28318 |
| *ex7.mtx* | 38554 | 34954 | 29174 | 25054 |
| *ex33.mtx* | 7408 | 8934 | 4920 | 5572 |
| *coater1.mtx* | 11722 | 11558 | 7684 | 7448 |
| *pesa.mtx* | 36972 | 41154 | 31010 | 33094 |

| Matrix (LFO) | $\|R_p\|$ | | $\|R_a\|$ | |
|---|---|---|---|---|
| | Algorithm 3.1 | Algorithm 3.2 | Algorithm 3.1 | Algorithm 3.2 |
| *steam1.mtx* | 64 | 1048 | 64 | 666 |
| *steam2.mtx* | 240 | 2624 | 240 | 1248 |
| *nos3.mtx* | 1880 | 6882 | 1246 | 4442 |
| *crystm01.mtx* | 20326 | 36634 | 12256 | 21194 |
| *ex7.mtx* | 37080 | 33426 | 28904 | 24060 |
| *ex33.mtx* | 10574 | 10564 | 7170 | 6888 |
| *coater1.mtx* | 11312 | 11512 | 7410 | 7536 |
| *pesa.mtx* | 42490 | 41676 | 31790 | 31884 |

| Matrix (SLO) | $\|R_p\|$ | | $\|R_a\|$ | |
|---|---|---|---|---|
| | Algorithm 3.1 | Algorithm 3.2 | Algorithm 3.1 | Algorithm 3.2 |
| *steam1.mtx* | 64 | 1294 | 64 | 754 |
| *steam2.mtx* | 240 | 3192 | 240 | 1912 |
| *nos3.mtx* | 1682 | 6772 | 1132 | 4382 |
| *crystm01.mtx* | 24478 | 45166 | 14252 | 26782 |
| *ex7.mtx* | 36486 | 34448 | 27044 | 24164 |
| *ex33.mtx* | 8024 | 10754 | 5186 | 7138 |
| *coater1.mtx* | 10476 | 11702 | 7004 | 7878 |
| *pesa.mtx* | 39606 | 44624 | 29034 | 34044 |

Table 3.2: The comparison between the number of potentially and additionally required elements computed with Algorithm 3.1 and Algorithm 3.2. The block size is fixed to 10. The orderings for coloring are (Top) the natural ordering, (Middle) LFO, and (Bottom) SLO.

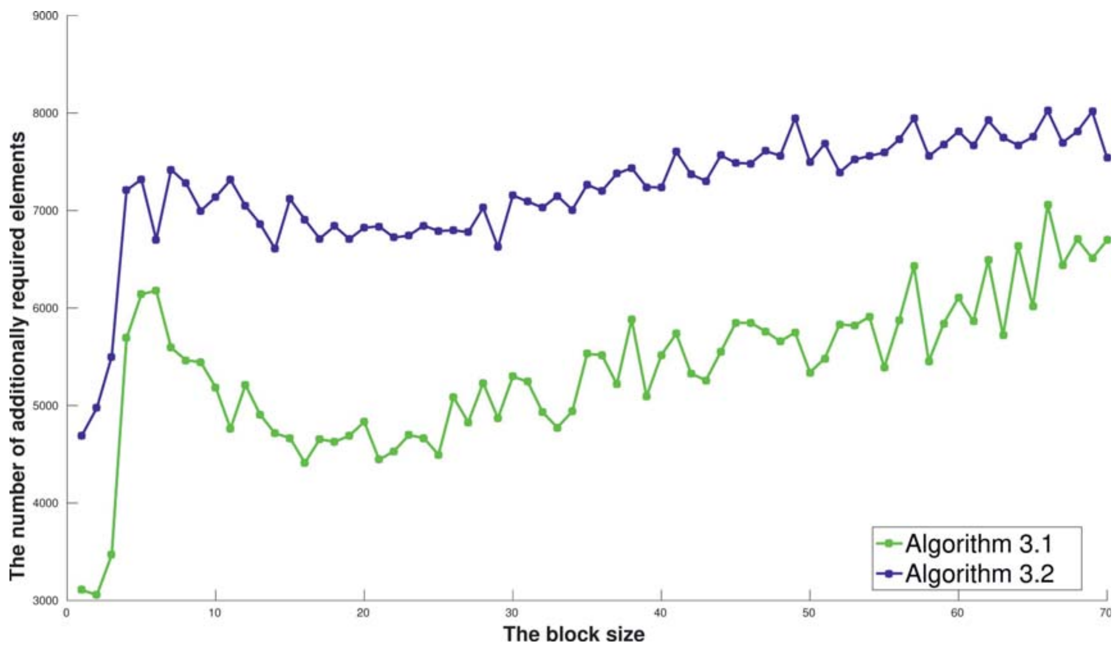


Figure 3.2: The number of additionally required elements computed by Algorithm 3.2 with the SLO ordering compared with Algorithm 3.1. The computation is carried out on the matrix *ex33*.

Algorithm 3.2 tends to perform better for smaller block sizes up to around 40. However, for larger block sizes, Algorithm 3.1 tends to produce slightly more additionally required elements. Figure Figure 3.5 shows that the number of colors does not vary significantly.

Now, we modify Algorithm 3.2 further to select a vertex with the minimum number of required elements among the computed set of vertices with the maximum number of determined nonrequired elements. This idea facilitates gathering of more nonrequired elements in the same column since more zero elements remain. These elements might offer more options for increasing the number of determined nonrequired elements. Let the function $M : V_c \to \mathbb{N}$ compute the number of required edges adjacent to a vertex. A new algorithm is proposed in Algorithm 3.4 which applies this idea. The only new difference of this algorithm to Algorithm 3.2 is to compute the operator $\arg\min$ for the function $M$. This computation has the same complexity as the computation of $\arg\max$. It means the time complexity is still as before. Figure 3.6 shows how the number of additionally required elements is increased comparing Algorithm 3.2 and Algorithm 3.4 when the ordering for the coloring is the LFO ordering. However, this is not the same for all matrices. Appendix A.2 shows the results for different matrices and different orderings for the block size fixed to 10. Except the matrix *ex7*, a general observation is that the numbers of additionally and potentially required elements are decreased when the number of colors is decreased. This brings us to the next topic of balancing the number of colors and the number of additionally required elements.

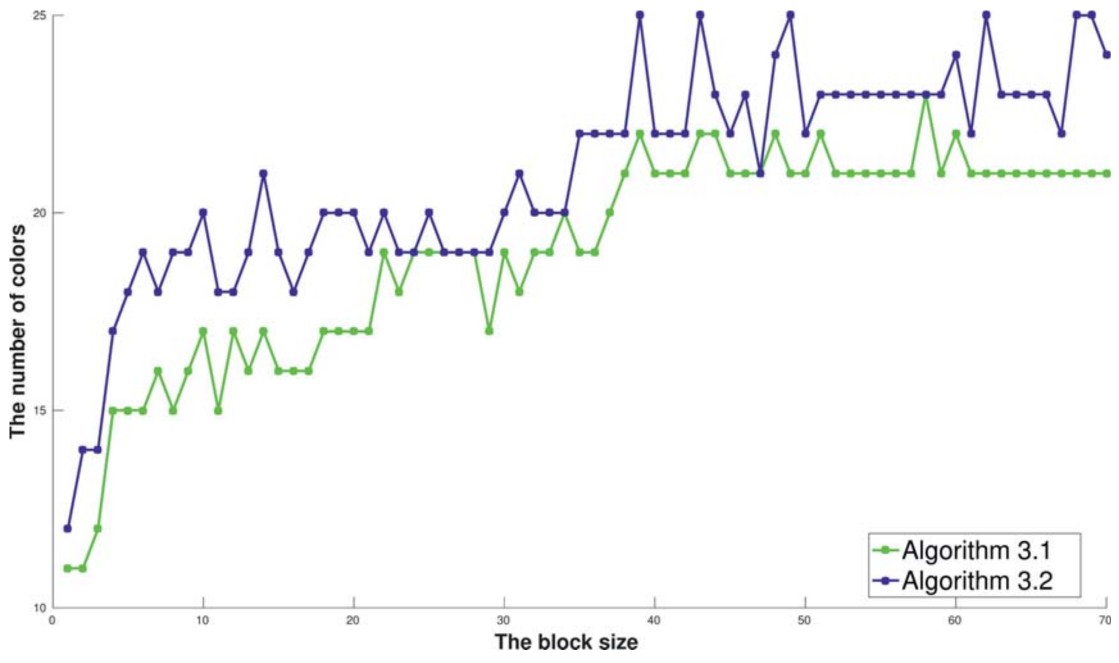


Figure 3.3: The number of colors computed by Algorithm 3.2 with the SLO ordering compared with Algorithm 3.1. The computation is carried out on the matrix *ex33*.

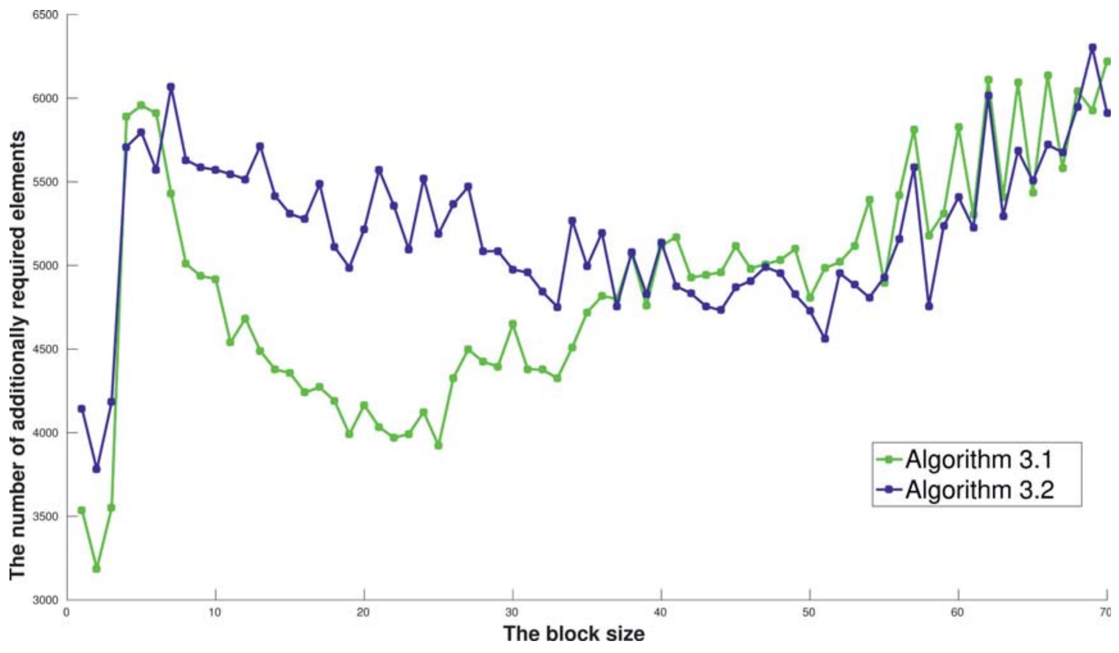


Figure 3.4: The number of additionally required elements computed by Algorithm 3.2 with the natural ordering compared with Algorithm 3.1. The computation is carried out on the matrix *ex33*.

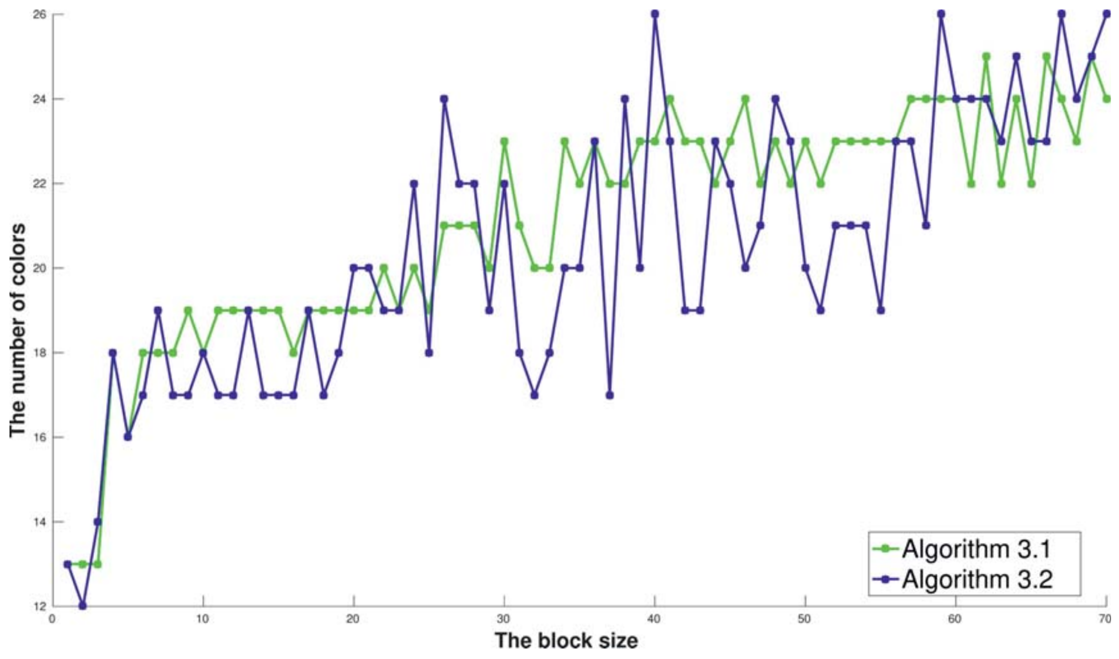


Figure 3.5: The number of colors computed by Algorithm 3.2 with the natural ordering compared with Algorithm 3.1. The computation is carried out on the matrix *ex33*.

```
function d2_color_nreq_modified($G = (V_r \cup V_c, E), E_i \subseteq E$)
  $\Phi \leftarrow [0 \dots 0]$
  $forbiddenColors \leftarrow [0 \dots 0]$
  for $v \in V_c$ with $\exists r \in V_r : (v, r) \in E_i$ and $\Phi(v) = 0$
    for $n \in N_2(v, E_i)$ with $\Phi(n) \neq 0$
      $forbiddenColors[\Phi(n)] = v$
    $\Phi(v) = \min\{a > 0 : forbiddenColors[a] \neq v\}$

    $I_v = \{z \in V_c : z \neq v$ and $z \notin N_2(v)$ and $\Phi(z) = 0\}$
    if $I_v \neq \emptyset$
      $maxs = \arg\max_{x \in I_v} L_v(x)$
      $mins = \arg\min_{x \in maxs} M(x)$
      $\Phi(mins[0]) = \Phi(v)$
  return $\Phi$
```

Algorithm 3.4: A modification of Algorithm 3.2.

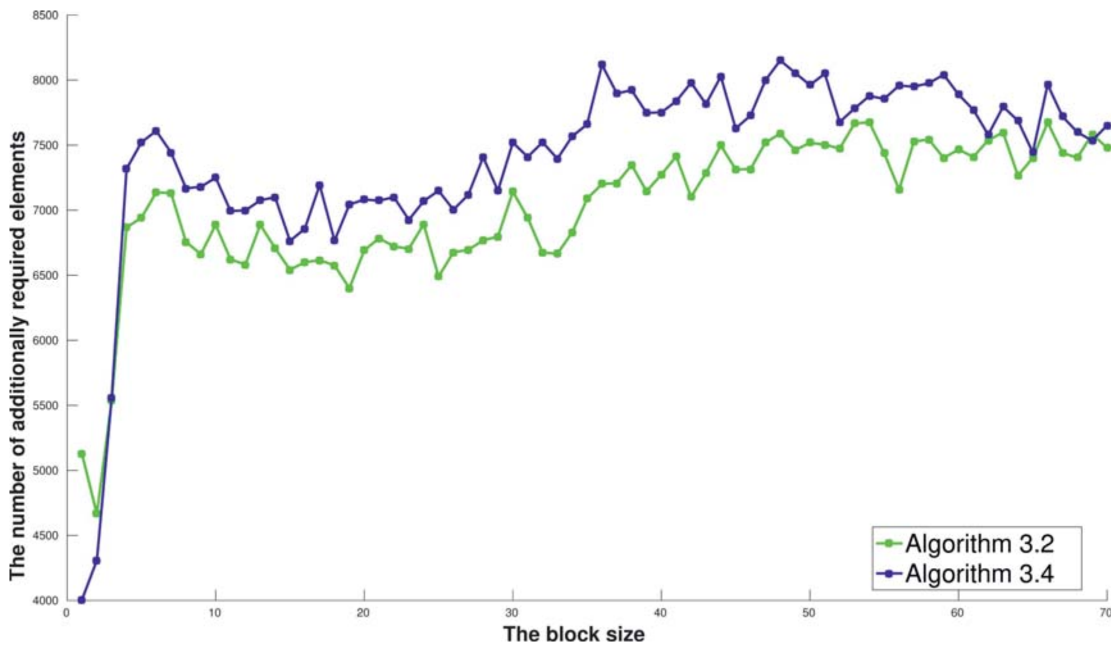


Figure 3.6: The number of additionally required elements is increased comparing Algorithm 3.2 and Algorithm 3.4 when the ordering for the coloring is the LFO ordering. The computation is carried out on the matrix *ex33*.

## Balancing the number of colors

Here, we modify Algorithm 3.2 and Algorithm 3.4 again to have a control over the balance between the number of colors and the number of additionally required elements. It means we define a balance variable $\alpha \in \mathbb{N}$ such that increasing this variable would decrease both the number of colors and additionally required elements and vice versa. Algorithm 3.5 presents the new algorithm. Rather than adding a single vertex as in Algorithm 3.4, the idea is to add more vertices representing columns with the minimum number of required elements and the maximum number of determined nonrequired elements. This is implemented by coloring all elements of $mins$ with indices from 0 to $\alpha - 1$ .

Figure 3.7 shows the comparison between the number of colors computed by the three presented algorithms: Algorithm 3.2, Algorithm 3.4, and Algorithm 3.5. All these algorithms are computed with the LFO ordering. We selected this ordering for this computation since this ordering gives the best results. The variable $\alpha$ for Algorithm 3.5 is equal to 10. The block size varies also from 1 to 70. It is clear from this figure that Algorithm 3.5 tends to generate the smallest number of colors particularly in the middle block sizes when $\alpha = 10$. However, as illustrated in Figure 3.8, it does not perform well in the case of additionally required elements.

Table 3.3 compares the number of additionally required elements and the number of colors using Algorithm 3.5 with different $\alpha$. Three tables at the top, middle, and the bottom of Table 3.3 are for the natural ordering, the LFO ordering, and the SLO ordering,

```
function d2_color_nreq_balance($G = (V_r \cup V_c, E), E_i \subseteq E, \alpha$)
  $\Phi \leftarrow [0 \dots 0]$
  $forbiddenColors \leftarrow [0 \dots 0]$
  for $v \in V_c$ with $\exists r \in V_r : (v, r) \in E_i$ and $\Phi(v) = 0$
    for $n \in N_2(v, E_i)$ with $\Phi(n) \neq 0$
      $forbiddenColors[\Phi(n)] = v$
    $\Phi(v) = \min\{a > 0 : forbiddenColors[a] \neq v\}$

    $I_v = \{z \in V_c : z \neq v$ and $z \notin N_2(v)$ and $\Phi(z) = 0\}$
    if $I_v \neq \emptyset$
      $maxs = \arg\max_{x \in I_v} L_v(x)$
      $mins = \arg\min_{x \in maxs} M(x)$
      for $i \in \{0, 1, ..., \min(\alpha - 1, \text{size}(mins) - 1)\}$
        $\Phi(mins[i]) = \Phi(v)$
  return $\Phi$
```

Algorithm 3.5: New coloring heuristic with a controller to balance the number of colors and the number of additionally required elements.

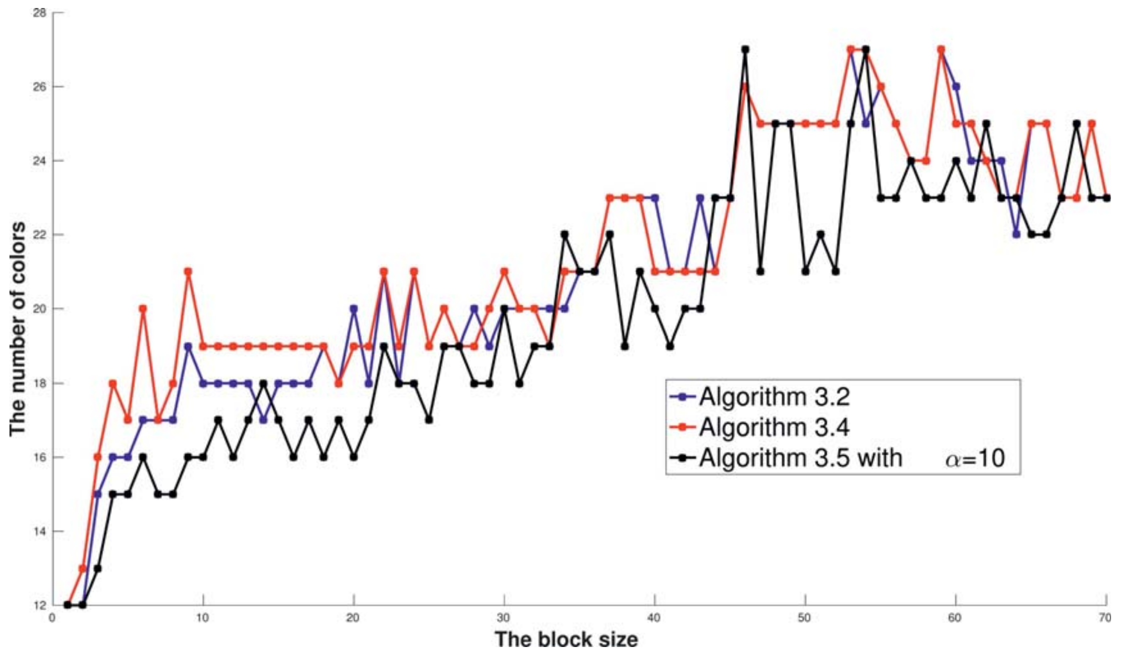


Figure 3.7: The comparison of the number of colors in Algorithm 3.2, Algorithm 3.4, and Algorithm 3.5. The computation is carried out on the matrix *ex33* and with the LFO ordering.

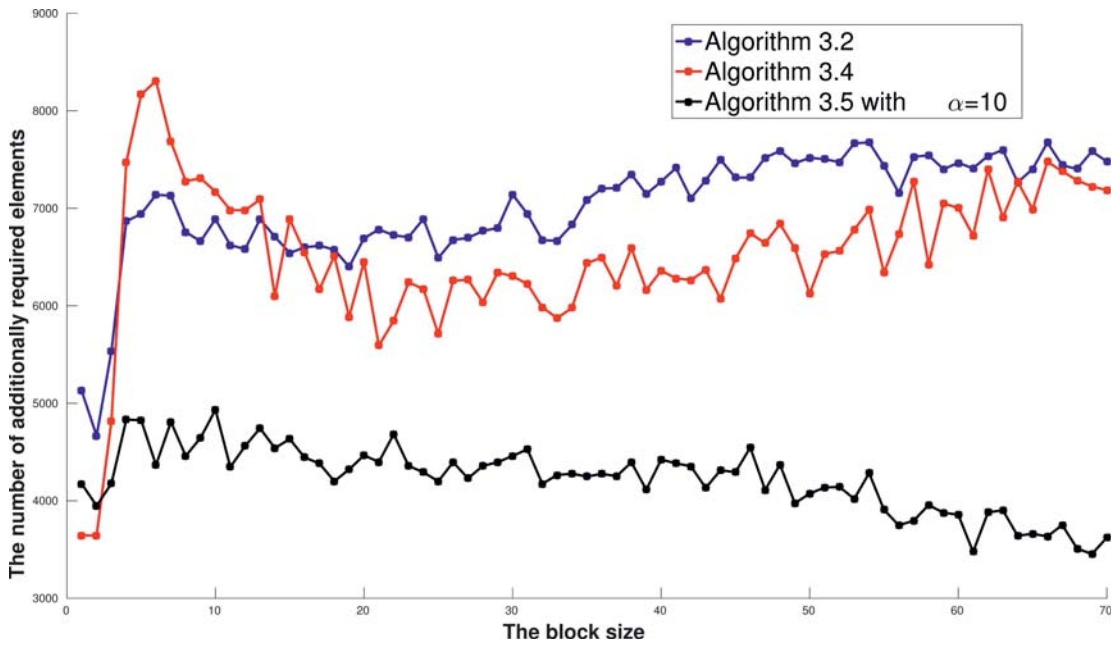


Figure 3.8: The comparison of the number of additionally required elements in Algorithm 3.2, Algorithm 3.4, and Algorithm 3.5. The computation is carried out on the matrix *ex33* and with the LFO ordering.

respectively. Regardless of the ordering and except for certain cases, the number of colors tends to decrease when $\alpha$ grows. Indeed, the number of colors is reduced in most of the cases. In some cases like the matrix *steam2.mtx* with the LFO ordering, we need 6 fewer colors comparing the coloring with $\alpha = 0$ and $\alpha = 10$.

On the other hand, except for certain cases, the number of additionally required elements decreases when $\alpha$ increases in the coloring. However, in this table, the coloring with $\alpha = 6$ can be a suitable choice in most of the cases since the number of additionally required elements is relatively high while the number of colors is mainly small. More figures can be found in Appendix A.3 comparing these values based on varying block sizes.

| Matrix (NAT) | Algorithm 3.5 | | | | | |
|---|---|---|---|---|---|---|
| | $\|R_a\|$ | | | $\|\Phi\|$ | | |
| | $\alpha = 0$ | $\alpha = 6$ | $\alpha = 10$ | $\alpha = 0$ | $\alpha = 6$ | $\alpha = 10$ |
| *steam1.mtx* | 566 | 440 | 370 | 10 | 9 | 8 |
| *steam2.mtx* | 1512 | 944 | 1032 | 13 | 10 | 9 |
| *nos3.mtx* | 3416 | 2778 | 2348 | 20 | 18 | 17 |
| *ex7.mtx* | 23958 | 22656 | 21180 | 55 | 49 | 46 |
| *ex33.mtx* | 5992 | 5616 | 5262 | 19 | 18 | 18 |
| *crystm01.mtx* | 28348 | 28466 | 28516 | 22 | 24 | 24 |
| *coater1.mtx* | 7896 | 7562 | 7538 | 30 | 26 | 25 |

| Matrix (LFO) | Algorithm 3.5 | | | | | |
|---|---|---|---|---|---|---|
| | $\|R_a\|$ | | | $\|\Phi\|$ | | |
| | $\alpha = 0$ | $\alpha = 6$ | $\alpha = 10$ | $\alpha = 0$ | $\alpha = 6$ | $\alpha = 10$ |
| *steam1.mtx* | 518 | 456 | 330 | 12 | 10 | 10 |
| *steam2.mtx* | 1280 | 660 | 564 | 17 | 13 | 11 |
| *nos3.mtx* | 3646 | 2360 | 2190 | 20 | 19 | 19 |
| *ex7.mtx* | 22532 | 21444 | 21576 | 49 | 48 | 47 |
| *ex33.mtx* | 5968 | 5222 | 4934 | 16 | 17 | 16 |
| *crystm01.mtx* | 21168 | 21918 | 21210 | 18 | 18 | 18 |
| *coater1.mtx* | 7210 | 7168 | 6998 | 23 | 23 | 23 |

| Matrix (SLO) | Algorithm 3.5 | | | | | |
|---|---|---|---|---|---|---|
| | $\|R_a\|$ | | | $\|\Phi\|$ | | |
| | $\alpha = 0$ | $\alpha = 6$ | $\alpha = 10$ | $\alpha = 0$ | $\alpha = 6$ | $\alpha = 10$ |
| *steam1.mtx* | 748 | 476 | 390 | 14 | 13 | 13 |
| *steam2.mtx* | 1816 | 1024 | 980 | 17 | 12 | 11 |
| *nos3.mtx* | 3998 | 2620 | 1978 | 20 | 18 | 17 |
| *ex7.mtx* | 23598 | 22362 | 22098 | 52 | 50 | 49 |
| *ex33.mtx* | 6174 | 5752 | 4902 | 19 | 18 | 17 |
| *crystm01.mtx* | 27432 | 26478 | 27782 | 22 | 22 | 22 |
| *coater1.mtx* | 7668 | 7624 | 7570 | 25 | 26 | 24 |

Table 3.3: The comparison of the additionally required elements and the number of colors in the computation of Algorithm 3.5 with the different value of $\alpha$. The orderings for coloring are (Top) the natural ordering, (Middle) LFO, and (Bottom) SLO.

| OpenMP-parallelized Algorithm 3.1 | | |
|---|---|---|
| Threads | Time | Colors |
| 1 | 42.8745 | 47 |
| 2 | 33.9665 | 47 |
| 3 | 25.2741 | 48 |
| 4 | 20.6863 | 48 |
| 5 | 21.4943 | 47 |

| OpenMP-parallelized Algorithm 3.5 | | |
|---|---|---|
| Threads | Time | Colors |
| 1 | 96.795 | 48 |
| 2 | 75.744 | 47 |
| 3 | 55.605 | 49 |
| 4 | 49.335 | 49 |
| 5 | 49.360 | 47 |

Figure 3.9: (Left) The results of computation of OpenMP-parallelized Algorithm 3.1. (Right) The results of computation of OpenMP-parallelized Algorithm 3.5.

## Parallelization

For a faster computation, we parallelize the proposed heuristics by OpenMP. There is much literature to parallelize coloring algorithms. For example, Çatalyürek [34] introduced an OpenMP parallelized greedy coloring which computes the same number of colors as the serial version. However, there are two points in each iteration of the algorithm in which the threads need to be synchronized. Here, our focus is on another algorithm from Rokos et al. [35] which presents an algorithm in which only a single point of synchronization is needed. In this algorithm, the number of colors changes with the number of parallel threads but it remains near to the number of colors in the serial version. We adapt this parallelization to Algorithm 3.1 and Algorithm 3.5 for the bipartite graph model. These algorithms first color the vertices greedily with a parallel loop and then correct the false coloring which can happen.

We color the bipartite graph associated with the matrix *Cavity16* from the sparse matrix collection of the university of Florida. The timing results are all from the computations carried on an Intel Core i5 with 8 GB RAM with hyperthreading. This means that the number of available parallel threads is doubled by hyperthreading resulting in 4 threads. Table 3.9 shows the results of these computations. Here, we change the number of threads from 1 to 5 shown in the first column. The second column shows the computation time in milliseconds. The third column is also the number of colors which changes based on the number of threads. Here, the time decreases by increasing the number of threads up to 4. Then it decreases as we have only 4 available threads. Also, the number of colors changes slightly by increasing the number of threads.

### 3.1.2 Restricted star bicoloring

Here, we search for a modified version of star bicoloring which increases the number of potentially and additionally required elements without a high increase in the number of colors. We can apply the ideas from Algorithm 3.4 and Algorithm 3.5. However, we should not expect to have a high increase in the number of additionally required elements that is equal to the sum of the increases in the column compression and row compression.

In [11], there is an algorithm called *star bicoloring scheme*. Lülfesmann [13] implemented and evaluated this algorithm. This algorithm closely follows the definition of the star bicoloring. Complete direct cover bicoloring [7] is another algorithm which is introduced for bicoloring. According to Calotoiu [12], this algorithm does not perform better than the star bicoloring scheme. Also, Calotoiu [12] introduces two other algorithms the integrated star bicoloring and total ordering star bicoloring which perform better than the star bicoloring scheme for certain matrices and not much worse for other matrices.

We consider the implementation of star bicoloring in Algorithm 3.6 based on the algorithm from Lülfesmann [14]. Here, the notation $G[S]$ means a graph $G$ induced by the edge set $S$. Also, the notation $\Delta(V, G)$ represents the maximum degree of the vertices from $V$ in the graph $G$. In this algorithm, the function *next_vertex* selects first which vertex should be processed in the next step. Should it be from the column vertices or the row vertices? This selection is based on the value of a weightening factor $\rho$ (to be explained below) and a vertex with the maximum degree in each step. After this selection, the conditions of the star bicoloring are analyzed in the next lines of this algorithm. Two final loops of the algorithm are just post processing steps. The first one makes the colors distinct for the column and row vertices. And the second one colors the uncolored vertices with the color 0.

Before introducing our new heuristic, we first do computations to find the influence of $\rho$ on the number of additionally required elements. The value of $\rho$ is a weightening factor which is a balance between columns or row vertices. A higher value of $\rho$ makes the compression mostly in the column vertices and a smaller value of $\rho$ makes the compression mostly in rows. There are some discussion on how to choose the value of $\rho$ in [11]. Also, Lülfesmann [14, 13] did some computation for some specific $\rho$. However, the goal in the previous literature is to minimize the number of colors. As the Figure 3.10 and Figure 3.11 show, the value of $\rho$ has a direct influence on the number of additionally required elements. The interesting observation is that a tiny change in the value of $\rho$ can dramatically change the results. For example, changing the value of $\rho$ from 0.3 to 0.4 in Figure 3.11 would result in a big change in the number of additionally required elements.

Now, we introduce our new heuristic which is a modification of the function *next_vertex* of Algorithm 3.6. The other parts of Algorithm 3.6 remain the same as in our new heuristic. Algorithm 3.7 shows the new function *next_vertex_nreq* which should be called instead of *next_vertex* to get the next vertex in each step. The global variable $NonReq$ switches between two modes in the new function *next_vertex_nreq*. The first mode $NonReq = false$ selects the vertex based on the variable $\rho$ like Algorithm 3.6. The second mode $NonReq = true$ selects the next vertex with the maximum number of determined nonrequired elements

```
function star_bicoloring($G = (V_r \cup V_c, E)$, $E_i \subseteq E$, $\rho$, next_vertex)
  $\Phi \leftarrow [-1 \ldots -1]$
  $forbiddenColors \leftarrow [0 \ldots 0]$
  $E_i^{'} = E_i$
  while $E_i^{'} \neq \emptyset$
    $v$=next_vertex($G$, $E_i^{'}$, $\rho$)
    $E_i^{'} = E_i^{'} - \{(v,w) \in E_i^{'} : w \in N_1(v, G[E_i^{'}])\}$
    for $w \in N_1(v,G)$
      if $\Phi(w) \leq 0$
        for $x \in N_1(w,G)$ with $\Phi(x) > 0$
          if $(v,w) \in E_i$ or $(w,x) \in E_i$
            $forbiddenColors[\Phi(x)] = v$
      else
        for $x \in N_1(w, G[E_i])$ with $\Phi(x) > 0$
          for $y \in N_1(x,G)$ with $\Phi(y) > 0$ and $y \neq w$
            if $\Phi(w) = \Phi(y)$
              $forbiddenColors[\Phi(x)] = v$

    $\Phi(v) = \min(\{j > 0 : forbiddenColors[j] \neq v\})$
  for $v_c \in V_c$ with $\Phi(v_c) > 0$
    $\Phi(v_c) = \Phi(v_c) + \max(\{\Phi(v_r) : v_r \in V_r\})$
  for $v \in V_r \cup V_c$ with $\Phi(v) = -1$
    $\Phi(v) = 0$

  return $\Phi$

function next_vertex($G = (V_r \cup V_c, E)$, $E_i^{'}$, $\rho$)
  if($\Delta(V_r, G[E_i^{'}]) > \rho\Delta(V_c, G[E_i^{'}])$)
    $v$ = $v_r \in V_r$ with maximum degree in $G[E_i^{'}]$
  else
    $v$ = $v_c \in V_c$ with maximum degree in $G[E_i^{'}]$
  return $v$
```

Algorithm 3.6: Algorithm 3.5 from Lülfesmann [14] for star bicoloring.

```
function star_bicoloring_nreq($G = (V_r \cup V_c, E)$, $E_i \subseteq E$, $\rho$)
  NonReq = false
  star_bicoloring($G$, $E_i$, $\rho$, next_vertex_nreq)

function next_vertex_nreq($G = (V_r \cup V_c, E)$, $E_i^{'}$, $\rho$)
  $v$ = next_vertex($G$, $E_i^{'}$, $\rho$)
  if NonReq = false
    NonReq = true
  else
    $I_v = \{z \in V_c : z \neq v$ and $z \notin N_2(v)$ and $\Phi(z) = 0\}$
    if $I_v \neq \emptyset$
      $maxs = \arg\max_{x \in I_v} L_v(x)$
      $mins = \arg\min_{x \in maxs} M(x)$
      $v = mins[0]$
    NonReq = false
  return $v$
```

Algorithm 3.7: Our new heuristic for star bicoloring considering the determined nonrequired elements.

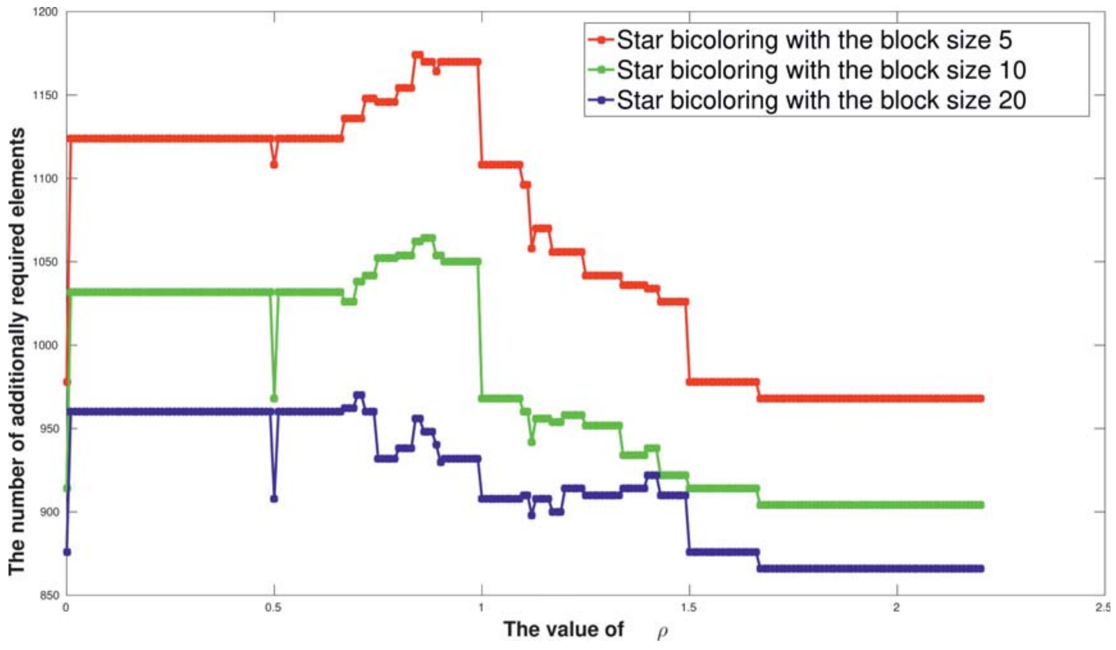


Figure 3.10: The influence of $\rho$ is computed on the additionally required elements with Algorithm 3.6 for the matrix *685_bus* with the LFO ordering.

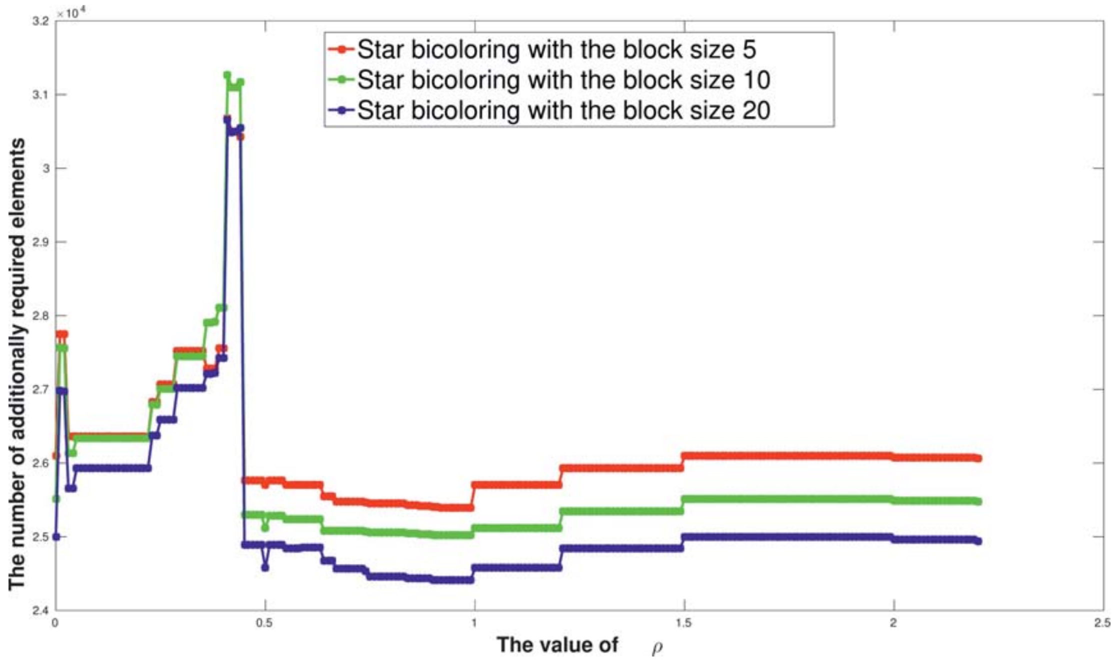


Figure 3.11: The influence of $\rho$ is computed on the additionally required elements with Algorithm 3.6 for the matrix *orani678* with the LFO ordering.

and the minimum number of required elements. Consider that $v$ is always computed in the first line of the function *next_vertex_nreq* but it will be modified later when $NonReq$ is equal to $true$.

Table 3.4 shows the numbers of potentially and additionally required elements computed with Algorithm 3.6 and Algorithm 3.7. Except for the matrix *coater01* with the natural ordering, we have an overall increase in both potentially and additionally required elements using Algorithm 3.7. Also, Figure 3.12 and Figure 3.13 represent the computation of Algorithm 3.6 and Algorithm 3.7 while the size of blocks is changing from 1 to 70. Again, an overall increase in the number of additionally required elements is achieved while the number of colors does not increase dramatically.

| Matrix (NAT) | $\|R_p\|$ | | $\|R_a\|$ | |
|---|---|---|---|---|
| | Algorithm 3.6 | Algorithm 3.7 | Algorithm 3.6 | Algorithm 3.7 |
| *steam1.mtx* | 64 | 590 | 64 | 454 |
| *steam2.mtx* | 240 | 2352 | 240 | 1648 |
| *nos3.mtx* | 4590 | 4614 | 2986 | 3050 |
| *ex7.mtx* | 35690 | 36486 | 28028 | 28796 |
| *ex33.mtx* | 9282 | 11180 | 6220 | 7510 |
| *crystm01.mtx* | 19262 | 22716 | 11472 | 13978 |
| *coater1.mtx* | 14402 | 14442 | 8296 | 8262 |
| *pesa.mtx* | 40572 | 41460 | 32728 | 33956 |

| Matrix (LFO) | $\|R_p\|$ | | $\|R_a\|$ | |
|---|---|---|---|---|
| | Algorithm 3.6 | Algorithm 3.7 | Algorithm 3.6 | Algorithm 3.7 |
| *steam1.mtx* | 64 | 802 | 64 | 466 |
| *steam2.mtx* | 240 | 2352 | 240 | 944 |
| *nos3.mtx* | 4824 | 5166 | 3152 | 3444 |
| *ex7.mtx* | 36794 | 37012 | 28670 | 28942 |
| *ex33.mtx* | 11070 | 11426 | 7380 | 7708 |
| *crystm01.mtx* | 21420 | 22714 | 13012 | 13992 |
| *coater1.mtx* | 14422 | 14496 | 8204 | 8350 |
| *pesa.mtx* | 42758 | 42904 | 32272 | 34266 |

| Matrix (SLO) | $\|R_p\|$ | | $\|R_a\|$ | |
|---|---|---|---|---|
| | Algorithm 3.6 | Algorithm 3.7 | Algorithm 3.6 | Algorithm 3.7 |
| *steam1.mtx* | 64 | 824 | 64 | 616 |
| *steam2.mtx* | 240 | 2320 | 240 | 1616 |
| *nos3.mtx* | 4314 | 4760 | 2784 | 3102 |
| *ex7.mtx* | 35690 | 36450 | 27814 | 28568 |
| *ex33.mtx* | 9728 | 10978 | 6468 | 7296 |
| *crystm01.mtx* | 24222 | 27226 | 14562 | 16590 |
| *coater1.mtx* | 14532 | 14634 | 8194 | 8412 |
| *pesa.mtx* | 41128 | 42112 | 31114 | 33744 |

Table 3.4: The comparison between the number of potentially and additionally required elements computed with Algorithm 3.6 and Algorithm 3.7. The block size is fixed to 10. The orderings for coloring are (Top) the natural ordering, (Middle) LFO, and (Bottom) SLO.

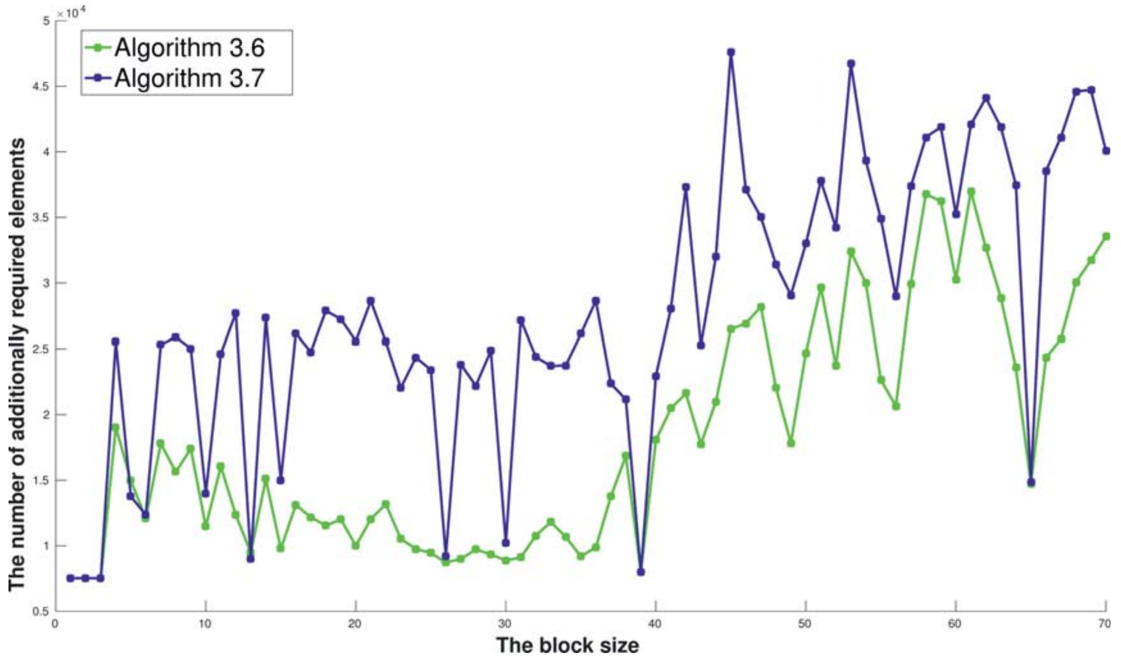


Figure 3.12: The number of additionally required elements computed with Algorithm 3.7 compared with Algorithm 3.6. The nonsymmetric matrix *crystm*01 with the natural ordering is used here.

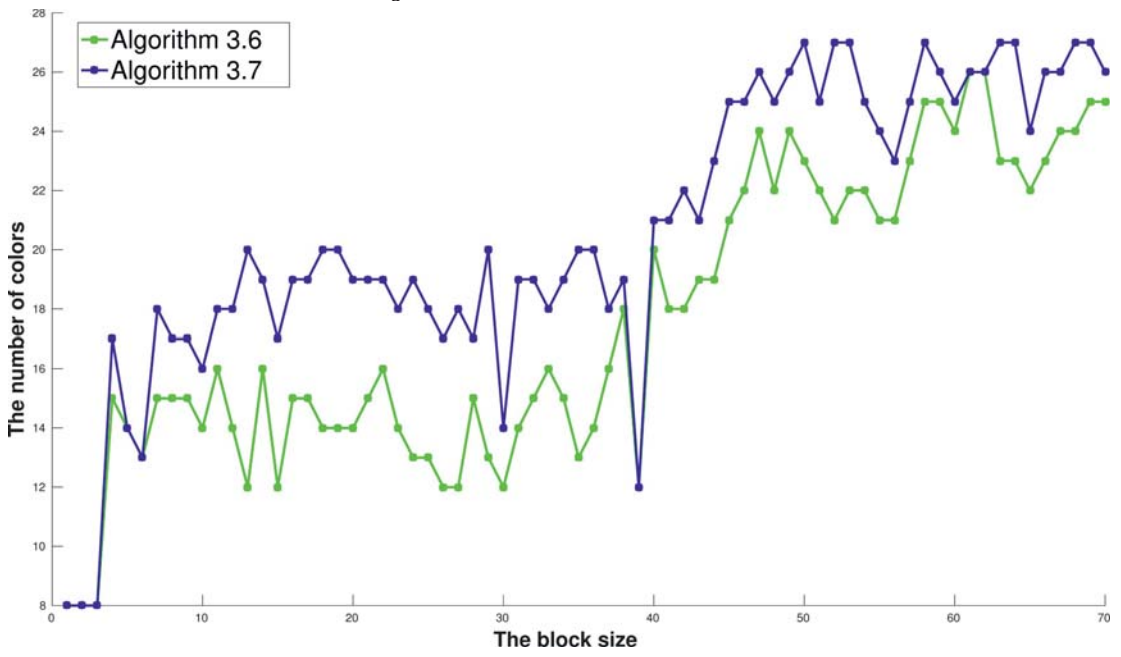


Figure 3.13: The number of colors computed with Algorithm 3.7 compared with Algorithm 3.6. The nonsymmetric matrix *crystm*01 with the natural ordering is used here.

## 3.2 Application in geoscience

Here, we apply our new heuristics to a carbon sequestration example from geoscience. The geophysics group of RWTH Aachen simulates the injection of $CO_2$ in a reservoir by a two-phase flow model in porous media. These two phases are a wetting and non-wetting phase like water and gas. A 2D two-phase flow can be formulated as a system of coupled nonlinear partial differential equations in which the boundary conditions are Dirichlet and Neumann. Integrating the time with the implicit Euler method based on [36, 14] results in the following system of nonlinear equations,

$$F(u) = 0 \text{ with } u = \begin{pmatrix} p_w \\ S_n \end{pmatrix} \in \mathbb{R}^{2MN} \text{ and } F = \begin{pmatrix} F_1 \\ F_2 \end{pmatrix} \in \mathbb{R}^{2MN},$$

where the variable $p_w \in \mathbb{R}^{MN}$ is the pressure for the wetting phase and $S_n \in \mathbb{R}^{MN}$ is the non-wetting saturation. The Newton's method solves this system of equations. Hence, the $2MN \times 2MN$ Jacobian matrix

$$A = \left(\begin{array}{c|c} \dfrac{\partial F_1}{\partial p_w} & \dfrac{\partial F_1}{\partial S_n} \\ \hline \dfrac{\partial F_2}{\partial p_w} & \dfrac{\partial F_2}{\partial S_n} \end{array}\right).$$

is determined with an AD-transformed version of the original function $F$. The AD tool ADiMat [37, 38] is used for this transformation.

This Jacobian matrix is divided into four quadrants: the derivative $\partial F_1/\partial p_w$ in the top left quadrant, $\partial F_1/\partial S_n$ in the top right, $\partial F_2/\partial p_w$ in the bottom left, and $\partial F_2/\partial S_n$ in the bottom right. Arising from a particular discretization, the sparsity patterns of the $462 \times 462$ Jacobian matrix $A$ with $3,236$ nonzeros is depicted in Figure 3.14. The discretization uses different stencils resulting in different sparsity patterns of quadrants. The Jacobian matrix

$$A = \left(\begin{array}{c|c} \text{five-point stencil} & \text{two-point stencil} \\ \hline \text{five-point stencil} & \text{five-point stencil} \end{array}\right) .$$

is based on the five-point stencil in the north west, south east, and south west quadrant. In the north east, the two-point stencil $\{(m, n), (m, n-1)\}$ with the center $(m, n)$ is used. More details are found in [14].

We solve the resulting systems of linear equations with the coefficient matrix $A$ in MATLAB using the BICGSTAB iterative solver. The right-hand side is the sum of all columns and the initial guess is the zero vector. We compute the ILU(2) preconditioning on a set of selected elements $S$ instead of the whole Jacobian matrix. We compare the convergence histories of the BICGSTAB solver on the four different methods of preconditioning: without preconditioning, preconditioning with $S = R_i$, preconditioning with $S = R_i \cup R_a$ in which the set $R_a$ is computed based on the greedy coloring in Algorithm 3.1,

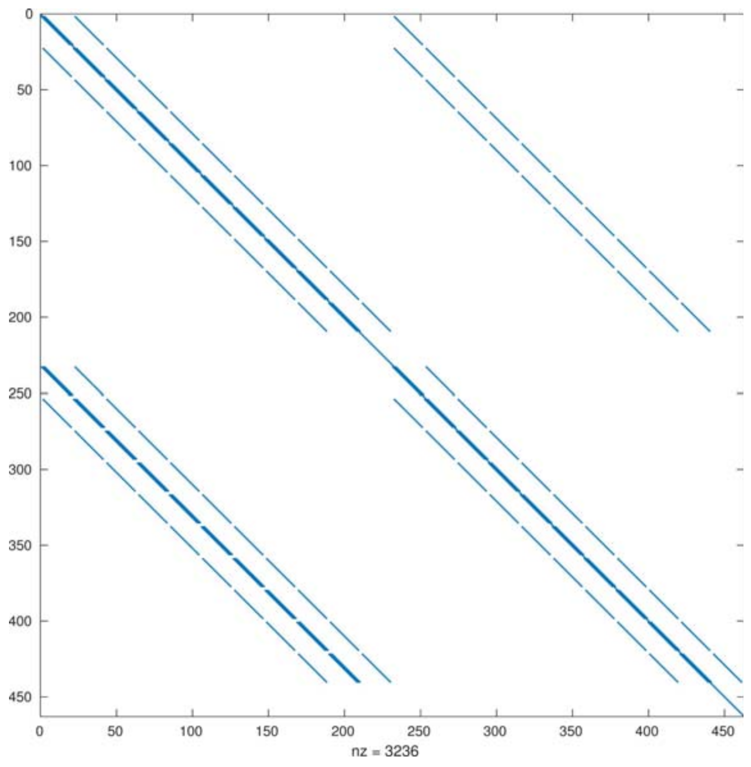
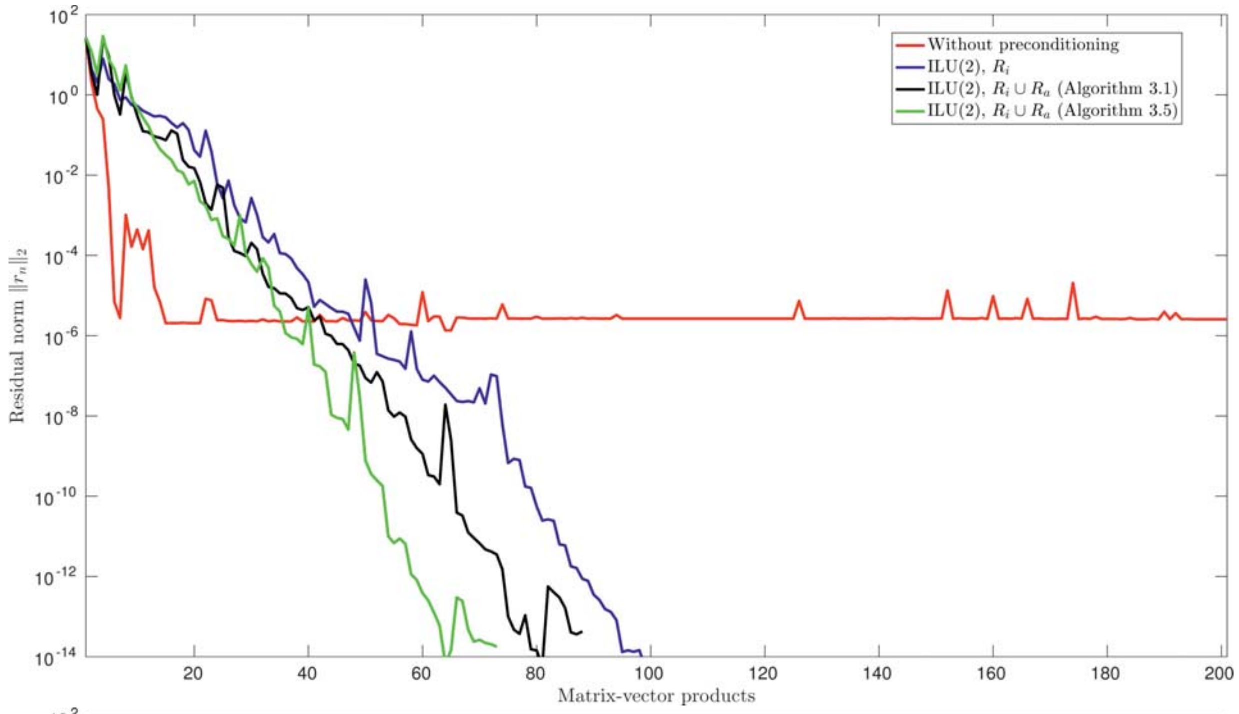


Figure 3.14: Sparsity pattern of Jacobian matrix $A$.

and preconditioning with $S = R_i \cup R_a$ in which the set $R_a$ is computed based on our proposed modified greedy coloring in Algorithm 3.5. In this section, Algorithm 3.5 is always computed with $\alpha = 6$.

We do this comparison for the four block sizes 5,10,15, and 20 in Figure 3.15 and Figure 3.16. In all of these block sizes, the preconditioning, which is computed based on the new coloring, converges in smaller numbers of matrix-vector products. However, it does not mean that a bigger block size results always in a better convergence as you can compare the figures for the block sizes 10 and 20.

Table 3.5 shows the results of the computation of Algorithm 3.1 and Algorithm 3.5 for the different block sizes and for the Jacobian $A$. The number of potentially and additionally required elements once more increases in all block sizes while the number of colors tends to be the same.



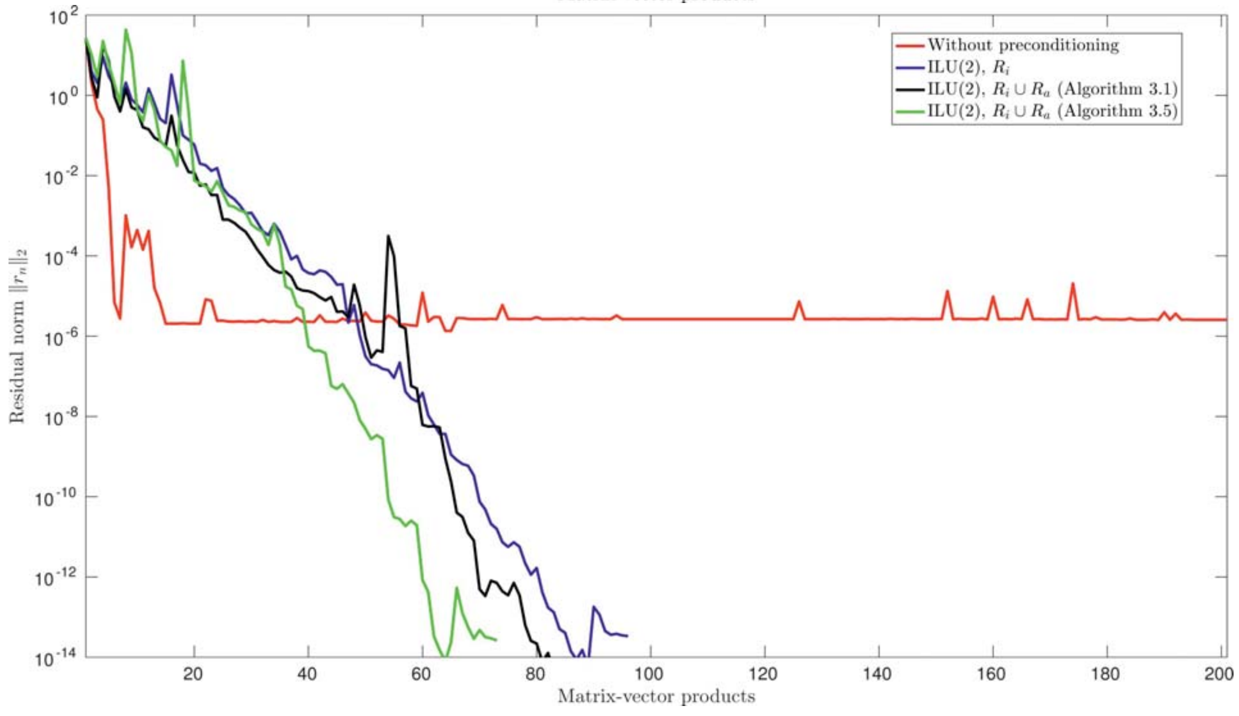
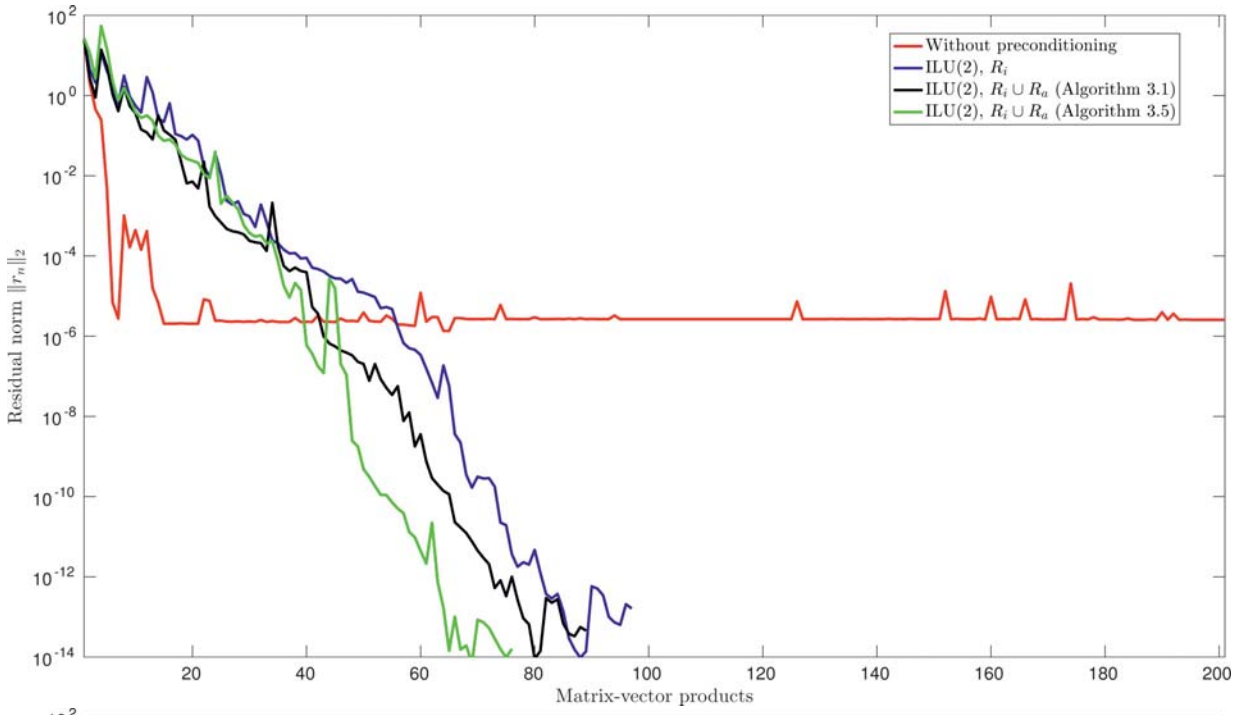


Figure 3.15: The history of the residual norms is visualized based on the number of matrix-vector products comparing four different methods: without preconditioning (red color), preconditioning with $S = R_i$ (blue color), preconditioning with $S = R_i \cup R_a$ in which the set $R_a$ is computed based on the greedy coloring (black color), and preconditioning with $S = R_i \cup R_a$ in which the set $R_a$ is computed based on Algorithm 3.5 (green color). The level parameter of the ILU preconditioning is 2. (Top) The block size is 5. (Bottom) The block size is 10.



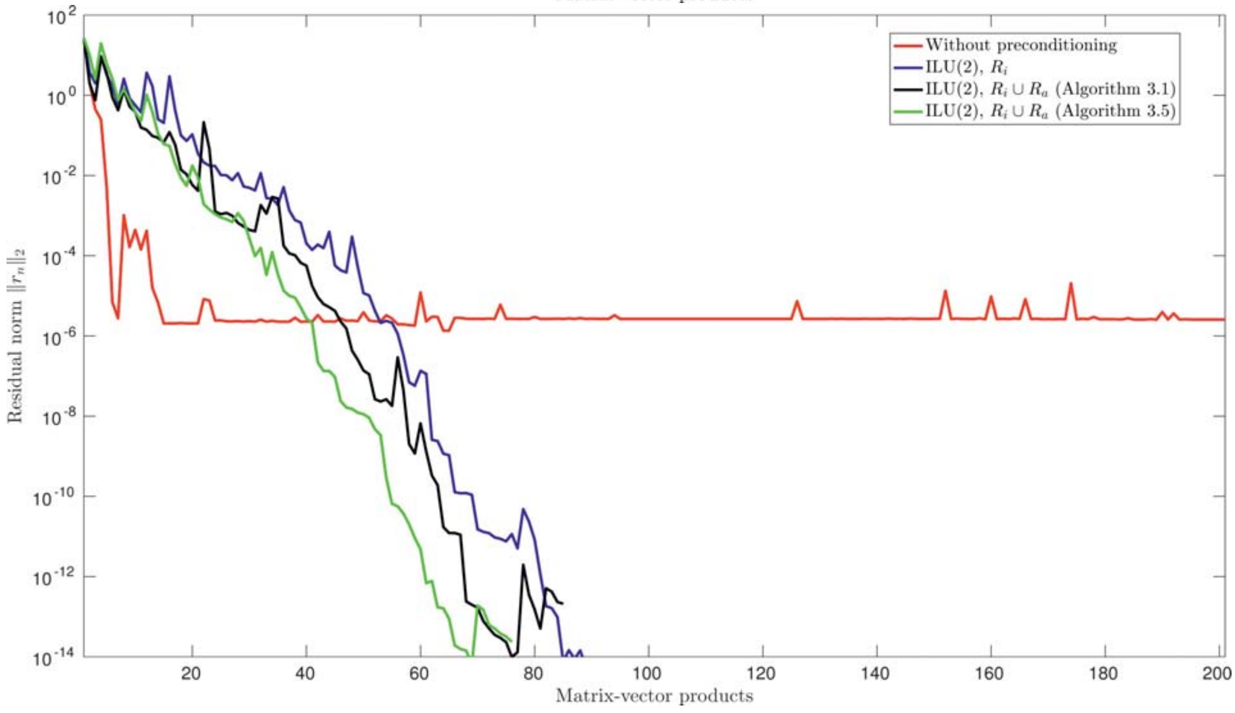


Figure 3.16: The history of the residual norms is visualized based on the number of matrix-vector products comparing four different methods: without preconditioning (red color), preconditioning with $S = R_i$ (blue color), preconditioning with $S = R_i \cup R_a$ in which the set $R_a$ is computed based on the greedy coloring (black color), and preconditioning with $S = R_i \cup R_a$ in which the set $R_a$ is computed based on Algorithm 3.5 (green color). The level parameter of the ILU preconditioning is 2. (Top) The block size is 15. (Bottom) The block size is 20.

| Size | Algorithm 3.1 | | | Algorithm 3.5 | | |
|---|---|---|---|---|---|---|
| | $\lvert R_p\rvert$ | $\lvert R_a\rvert$ | $\lvert\Phi\rvert$ | $\lvert R_p\rvert$ | $\lvert R_a\rvert$ | $\lvert\Phi\rvert$ |
| 1 | 734 | 724 | 6 | 1212 | 786 | 7 |
| 2 | 1670 | 886 | 11 | 1730 | 990 | 10 |
| 3 | 1316 | 776 | 10 | 1952 | 1052 | 14 |
| 4 | 1838 | 1016 | 13 | 1952 | 1070 | 14 |
| 5 | 1524 | 896 | 13 | 1976 | 1070 | 15 |
| 6 | 1456 | 732 | 13 | 1828 | 974 | 13 |
| 7 | 1596 | 852 | 11 | 1770 | 942 | 13 |
| 8 | 1758 | 928 | 13 | 1880 | 976 | 13 |
| 9 | 1750 | 912 | 13 | 1826 | 966 | 13 |
| 10 | 1704 | 960 | 13 | 1826 | 992 | 13 |
| 11 | 1744 | 904 | 13 | 1832 | 940 | 13 |
| 12 | 1738 | 900 | 13 | 1760 | 938 | 13 |
| 13 | 1712 | 874 | 13 | 1918 | 1022 | 13 |
| 14 | 1572 | 782 | 13 | 1572 | 848 | 13 |
| 15 | 1728 | 914 | 13 | 1836 | 928 | 13 |
| 16 | 1704 | 870 | 13 | 1842 | 956 | 13 |
| 17 | 1702 | 866 | 13 | 1812 | 940 | 14 |
| 18 | 1696 | 858 | 13 | 1704 | 874 | 13 |
| 19 | 1694 | 860 | 13 | 1784 | 940 | 13 |
| 20 | 1688 | 854 | 13 | 1810 | 956 | 13 |

Table 3.5: The comparison between the number of potentially and additionally required elements and the number of colors computed with Algorithm 3.1 and Algorithm 3.5. The computation is carried out on the matrix $A$. The block size is from 1 to 20. The ordering is the natural ordering.

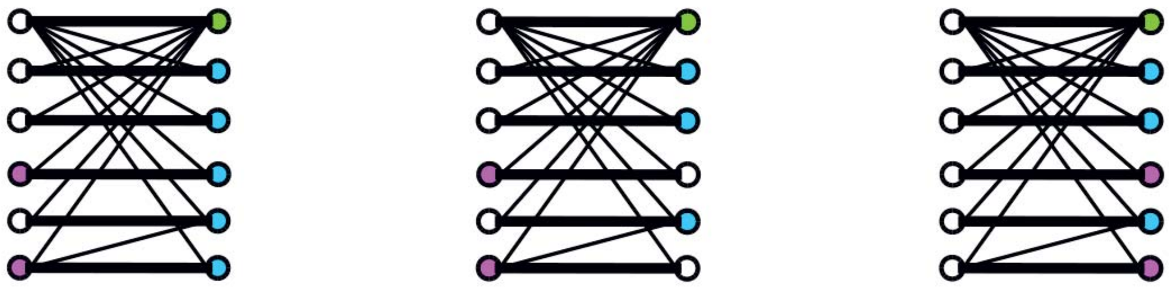


Figure 3.17: (Right) An example of a star bicoloring restricted to diagonal elements transformed to a valid distance-2 coloring.

## 3.3 Coloring restricted to diagonal elements

Here, we look at a particular example in which the sparsity pattern is restricted to diagonal elements. Lülfesmann [14] shows that the minimum number of colors of the star bicoloring restricted to diagonal elements $\chi_{sb}$ is equal to the minimal number of colors of the distance-2 coloring restricted to diagonal elements $\chi_{d2}$. It means that the bidirectional compression restricting to diagonals is not better than a distance-2 coloring restricted to diagonal elements. This theorem can be written as two lemmas,

- Lemma 1: $\chi_{sb} \geq \chi_{d2}$
- Lemma 2: $\chi_{sb} \leq \chi_{d2}$

The proof of Lemma 2 is clear since a given distance-2 coloring with the fewest number of colors is also a star bicoloring. The proof of Lemma 1 is more tricky. A sketch of the proof is as follow.

The idea is to transform a valid star bicoloring with the fewest number of colors $\Phi_{sb}$ to a valid distance-2 coloring where the number of colors is not larger. This transformation includes two steps:

1. The minimal star bicoloring $\Phi_{sb}$ is transformed to the valid star bicoloring $\Phi^*_{sb}$ where exactly one of both incident vertices of each required edge is colored nonzero.
2. The valid coloring $\Phi^*_{sb}$ is transformed to the valid star bicoloring $\Phi^{'}_{sb}$ where all row vertices are colored by zero. This star bicoloring is also a distance-2 coloring.

An example of this transformation can be seen in Figure 3.17. Looking at the proof of Lemma 3.8 of Lülfesmann [14], this lemma (and consequently Theorem 3.10 of Lülfesmann [14]) can be generalized by considering any coloring instead of the optimal coloring. Also, the set of diagonal elements in the theorem can be replaced by any set of required elements in which each column and row contain only one required nonzero element. This property is nothing but a matching [39] in a graph. Given a graph and the vertex and edge set $G = (V, E)$, a matching $M \subseteq E$ contains a list of edges which do not have any vertex in common. A maximum matching is a matching with a maximum possible number of edges. Then, the set of all diagonal elements for example in our bipartite graph is a maximum matching.

Now, we can formulate a new theorem on the comparison of unidirectional and bidirectional coloring as follows.

**Theorem 1** *Given the bipartite graph $G = (V_r \cup V_c, E)$ and a matching $M \subseteq E$ representing the required elements, any valid star bicoloring restricted to $M$ with the number of colors $X_{sb}$ can be transformed to a valid distance-2 coloring restricted to $M$ with the number of colors $X_{d2}$ such that $X_{sb} \geq X_{d2}$. These numbers of colors can be different from the minimal number of colors.*

This theorem is a generalization of Lemma 3.8 of [14]. This lemma discusses only the mapping with minimal coloring and the coloring restricted to diagonal elements. We consider any coloring restricted to a matching. The proof for that lemma can be applied to this new theorem considering the two following points,

- The minimality of colorings is only used to show the equality in the number of colors which is not the case in our theorem.
- The property of being diagonal elements is used in the proof from [14] only to make the required edges disjoint from each other which is the definition of a matching in a graph.

This generalization gives us an idea to find new heuristics for coloring in which we color the column and row vertices simultaneously. The motivation of this new heuristic is from the following observation. Next we compare the number of colors computed by Algorithm 3.1 and Algorithm 3.6 in Table 3.6 for different matrices. The better results are observed in Algorithm 3.6 for some matrices. These examples also show that the inequality in Theorem 1 can happen. The second column of this table shows the number of colors computed by Algorithm 3.6. The third column shows the minimum number of colors computed by the greedy coloring in Algorithm 3.1 either for rows or columns. It can be seen that even for a small matrix as the matrix *cage3* which has only 5 rows and columns, the star bicoloring can produce a better result.

More clearly, the idea is that a heuristic for star bicoloring maybe finds a better coloring than a distance-2 coloring because of the inequality in the theorem. As the proof of the theorem proposes, every required nonzero element can be determined by either a column or a row, but not both. Algorithm 3.8 shows our new heuristic. This heuristic iterates over the required edges. In each iteration, we simultaneously execute a distance-2 coloring for both incident vertices of the required edge. This greedy approach finds the minimum possible colors $c_1$ for columns and $c_2$ for rows, each in a separate fashion. Then, the recombination of these two separate distance-2 colorings into a star bicoloring is given by coloring the vertex with the corresponding smaller color. In numerical experiments we observe that the number of colors computed by this new heuristic is equal to $\Phi_{sb}$ for the matrices from Table 3.6.

| Matrix | $\Phi_{sb}$ | min($\Phi_r$,$\Phi_c$) |
|---|---|---|
| *cage3* | 3 | 4 |
| *cage4* | 3 | 4 |
| *cage5* | 5 | 7 |
| *cage7* | 7 | 8 |
| *cage8* | 8 | 10 |
| *cage9* | 9 | 11 |
| *cage10* | 10 | 11 |
| *cage12* | 13 | 14 |

| Matrix | $\Phi_{sb}$ | min($\Phi_r$, $\Phi_c$) |
|---|---|---|
| *ex7* | 18 | 22 |
| *nos3* | 10 | 10 |
| *steam1* | 6 | 6 |
| *steam2* | 8 | 8 |
| *rajat01* | 8 | 8 |
| *gyro_m* | 15 | 15 |
| *ex33* | 12 | 11 |
| *cavity16* | 20 | 18 |

Table 3.6: The comparison of the number of colors in star bicoloring restricted to diagonals $\Phi_{sb}$ computed by Algorithm 3.6 and in distance-2 coloring restricted to diagonals min($\Phi_r, \Phi_c$) (either for rows or for columns) computed by Algorithm 3.1. The ordering is the natural ordering of the matrix for both colorings.

```
function star_diag($G = (V_r \cup V_c, E)$, $E_i \subseteq E$)
  $\Phi \leftarrow [0 \dots 0]$
  $forbiddenColors1 \leftarrow [0 \dots 0]$
  $forbiddenColors2 \leftarrow [0 \dots 0]$

  for $e = (v, u) \in E_i$ with $v \in V_c$ and $u \in V_r$
    $forbiddenColors1[0] = v$
    $forbiddenColors2[0] = u$

    for $n \in N_2(v, E_i)$ with $\Phi(n) \neq 0$
        $forbiddenColors1[\Phi(n)] = v$

    for $n \in N_2(u, E_i)$ with $\Phi(n) \neq 0$
        $forbiddenColors2[\Phi(n)] = u$

    $c_1 = \min(\{j > 0 : forbiddenColors1[j] \neq v\})$
    $c_2 = \min(\{j > 0 : forbiddenColors2[j] \neq u\})$

    if $c_1 < c_2$
      $\Phi(v) = c_1$
    else
      $\Phi(u) = c_2$

  return $\Phi$
```

Algorithm 3.8: An improved star bicoloring restricted to diagonal elements. As the theorem says this coloring generates an equivalent distance-2 coloring.

## 3.4 Implementation details of PreCol

We develop a piece of software to implement our proposed heuristics. Specifically, the software is designed employing concepts from object-oriented programming such that it can be extended further with new coloring heuristics as well as preconditioning algorithms. The developers can implement new extensions without going into the details of the core implementation. Two main ingredients, coloring and orderings, can be implemented only by deriving an interface. For example, a new coloring and ordering can be added as easy as the following code.

```
class New_Ordering : public Ordering {
  void order(const Graph &G, vector<unsigned int> &ord, bool restricted)
     {...}
};

class New_Coloring : public ColAlg {
   vector<int> color() {...}
};
```

Here, the computed ordering is saved in the array *ord* and the coloring is the output of the function *color*.

The developer needs to implement these new classes in an only-header fashion [40] since the goal is to write an extension with little effort. So, the developer should move the new header file to the corresponding directory which is the *ordering* directory for this new ordering and the *algs* directory for coloring algorithms. Now, building the software will bring this new ordering into the software execution.

The input matrix is in the format of the matrix market [41]. After reading this matrix, we convert it to the different graph models like a column intersection graph or a bipartite graph. Any resulting matrix like the sparsified matrix will also be saved in this file format.

*PreCol* is developed in *C++* using STL (the standard library) and the boost library [42]. Using concepts of functional programming in the new C++ release (C++11 and C++14) [43], we provide different functions which can be used by a developer to work on graphs. For example, the iteration on vertices or edges can be as easy as follows.

```
for_each_v(G, f);
for_each_v(G, [&](Ver v) {...});
for_each_e(G, f);
for_each_e(G, [&](Edge e) {...});
```

in which the variable $f$ is a function which gets an input parameter of a vertex or an edge. Also, the other syntax is the lambda function from the new C++ functional programming to implement an unnamed function.

Following a unique solution, we implement all parts of our heuristics with the use of the standard library of *C++* which also improves the readability. This strategy reduces the code length dramatically. Also, the algorithms of the *C++* standard library will automatically be parallel in *C++17* [44].

As it is presented in Figure 3.18, the computations of coloring and ILU preconditioning

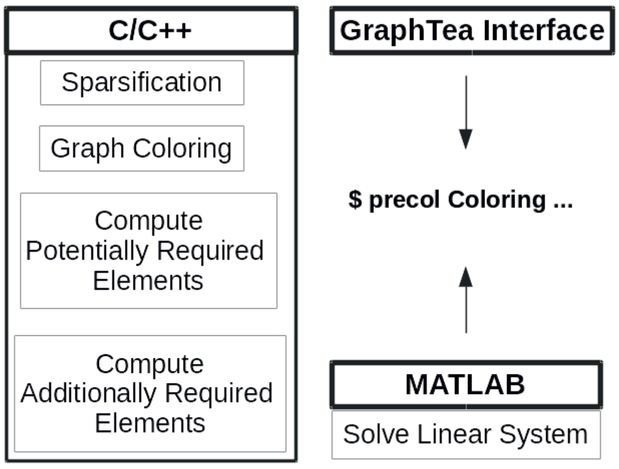


Figure 3.18: The software *PreCol* and two user interfaces written in MATLAB and Java.

are in one package available in *C++*. But, the computation of iterative solvers is carried out in MATLAB.

To use the software, the user can use a shell command. So, the user needs to specify different options for coloring algorithms, orderings, the block size, and so on. These options can be entered directly in the terminal. An example is as follows.

```
precol PartialD2RestrictedColumns LFO_Nat BlockDiagonal 30 2 ex33.mtx
```

in which the *PartialD2RestrictedColumns* is the coloring algorithm, the string *LFO_Nat* containing two strings *LFO* and *Nat* are for the coloring and ILU orderings. The next parameter *BlockDiagonal* specifies the sparsification method which is followed by the size of the block. Here, the block size is 30. The next number 2 specifies the level parameter of ILU. The matrix name is the last parameter.

We develop two user interfaces for *PreCol*. So that it can be called from within MATLAB and GraphTea [45, 46]. These user interfaces help to evaluate our proposed heuristics. Both interfaces execute the binaries of *PreCol* and process the output files generated by *PreCol*. The corresponding commands in both interfaces can be executed by the following parameters,

```
function (R_i,R_p,R_a,Φ) = precol(coloring,
  coloring_ordering,ilu_ordering,block_size,
  ILU_level_parameter,matrix_name,α)
...
```

in which the input parameters are passed directly to *PreCol*.

# 4 Interactive educational modules

We develop an extensible collection of educational modules (EXPLAIN) to teach combinatorial scientific computing in classroom. There is an increasing need for such educational tools since the connection between the problems from scientific computing and the corresponding combinatorial problem is tricky for the students. This chapter summarizes our previous publications [16, 17, 18, 19, 20] and also provides some new features.

Since graphs are ubiquitous in computer science, mathematics, and a variety of other scientific disciplines there are plenty of software tools for teaching graph-theoretical topics and graph algorithms. However, to the best of the authors' knowledge, there is no other software than EXPLAIN that provides the simultaneous visualization of a graph and a matrix next to each other. This overall layout of EXPLAIN is crucial to better understand the relationship between the graph problem and the corresponding matrix problem. These two different views of the same problem are critical for establishing an understanding of the problem at hand.

Though there is no previous work directly related to that area, we shortly mention the Gato/CATBox [47] software whose focus is on animation of graph algorithms. Similarly, the CABRI-Graph [48] software is mainly used for algorithm visualization. There are also many software tools in the area of information visualization like Tulip [49, 50] that visualize and analyze graphs. However, all these graph software tools do not involve any aspects of scientific computing. On the other hand, existing tools with a focus on scientific computing do not involve any aspects from graph theory. Examples include the interactive Java applets devoted to the textbook by Heath [51] and the NCM software to be used in conjunction with the textbook by Moler [52].

During this chapter, we first look at the concept and design of the software in Section 4.1. Then, we apply the gamification ideas on the software in Section 4.2 to involve the students more into the usage of EXPLAIN. After looking at the available modules in Section 4.3, we look at implementation details in Section 4.5. EXPLAIN has two releases 1.0 and 2.0 which we will explain in more detail in Section 4.5. Some of the modules are only available in the new release. Both releases are available now since they were developed on different technologies and have pros and cons.

## 4.1 Concept and design

Throughout the design stage of EXPLAIN, our focus is on the following goals. EXPLAIN is not a self-study tool. Rather, the connection between the scientific computing and combinatorial problems is explained by a teacher in classroom. The students can follow the algorithm on the graph by either clicking on the vertices or edges. So, neither the matrix nor its nonzero elements are clickable. Clicking vertices or edges results in modifications in both graph and matrix. The available modifications on the graph and the matrix can be one or more actions from the following list,

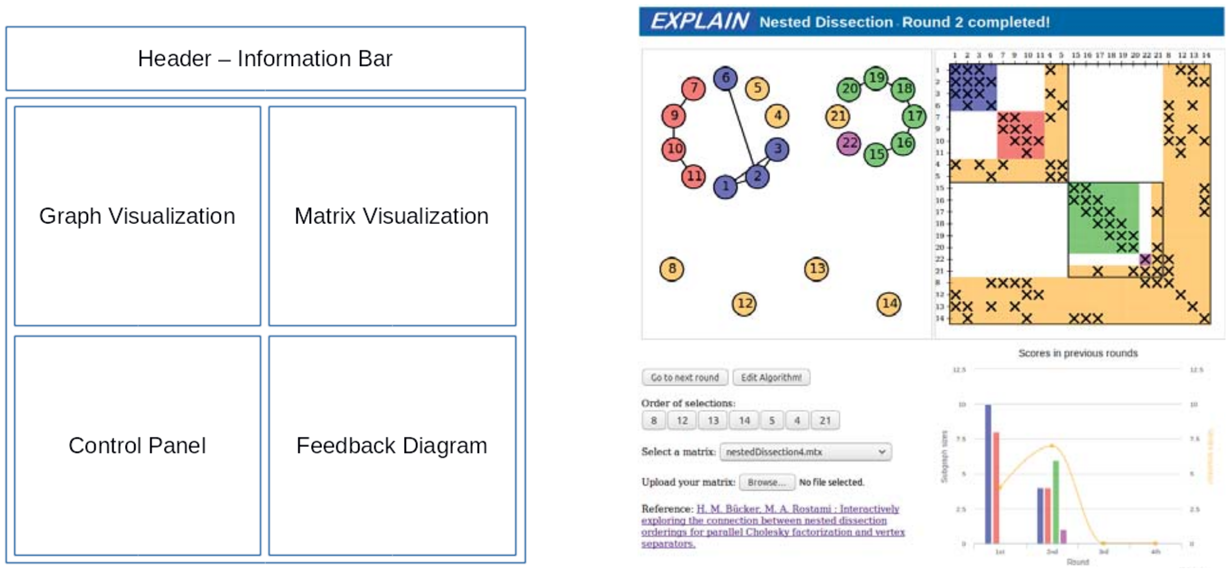
- Removing, adding, or coloring a vertex
- Removing, adding, or coloring an edge
- Changing the positions of vertices
- Permuting matrix columns or rows or both.
- Coloring any element, column, or row of the matrix

The input to the program is a sparse matrix in the format of the matrix market [41]. We build the corresponding graph from the matrix. The type of graph can be different based on the module and the algorithm. A list of possible graph types is as follows.

- Simple graph: an undirected graph without self-loops considering the given matrix with a symmetric pattern as an adjacency matrix of the graph.
- Directed graph: a directed graph without self-loops considering the given nonsymmetric matrix as an adjacency matrix of the graph.
- Column intersection graph: the graph model explained in Definition 2.
- Bipartite graph: the bipartite graph model explained in Definition 4.

The matrix and the corresponding graph are visualized side by side. Based on our goal to visualize the connection of the algorithm on the matrix and graph sides simultaneously, we design the software to have the four sections and a header as illustrated in Figure 4.1 (Left). The header contains the textual feedback to the user. For example, the completion of an algorithm is a textual feedback. The four sections are the graph view, the matrix view, the control panel, and the feedback diagram. The graph is drawn on the circular layout first. Other layouts can be selected later in the control panel. The matrix is visualized also at the right side and its nonzero elements are shown by the notation $\times$. Figure 4.1 (Right) shows an actual example of the implemented view of the nested dissection module.

We define a set of predefined colors which can be selected by the corresponding numbers, for example $\{1 = \text{green}, 2 = \text{turquoise}, 3 = \text{orange}, 4 = \text{violet}, 5 = \text{red}, 6 = \text{yellow}, ...\}$. These colors are selected such that they look perceptually distinct. However, a user can define any new color by a function that specifies an **rgb** value. Another aspect of the design is to use the same colors in the graph and matrix views as well as in the feedback diagram. This consistent use of colors in the graph view, the matrix view, and in the feedback diagram makes it easier for the student to understand the algorithm. We will see examples of this aspect in the different modules which we explain in Section 4.3.



Reference: H. M. Bücker, M. A. Rostami : Interactively exploring the connection between nested dissection orderings for parallel Cholesky factorization and vertex separators.

Figure 4.1: (Left) EXPLAIN has a fixed layout consisting of four sections and a header. (Right) An example of the actual implemented view of EXPLAIN.

## 4.2 Gamification

To engage the students more in the teaching process, we improved EXPLAIN such that the students get more feedback from the software. This concept is called gamification [53, 54]. The use of elements from game design in the context of computer science education is not new. In particular, programming assignments can involve implementations of games. In [55], for instance, an introductory programming course is taught under the common umbrella of two-dimensional game development. Similarly, a game project is used in a course on software architecture [56]. Programming assignments can also involve pieces of software that act as a player in an existing game. Rather than implementing a game, we are interested in situations where students learn by playing a game. A publication addressing this aspect of gamification is given in [57] where game-based learning is used to teach a course in data structures and algorithms. A collaborative game is described in [58] that aims at improving the teaching quality of a course on mathematical logic.

The gamification of EXPLAIN is based on finding a solution to a combinatorial scientific computing problem. We interpret each solution to a problem instance as a *round*. The feedback diagram reports the results of previous rounds. The idea of gamification is used to solve a combinatorial minimization problem. For example, the gamification in the nested dissection ordering consists of minimizing the size of the vertex separator while, at the same time, balancing the sizes of the remaining components.

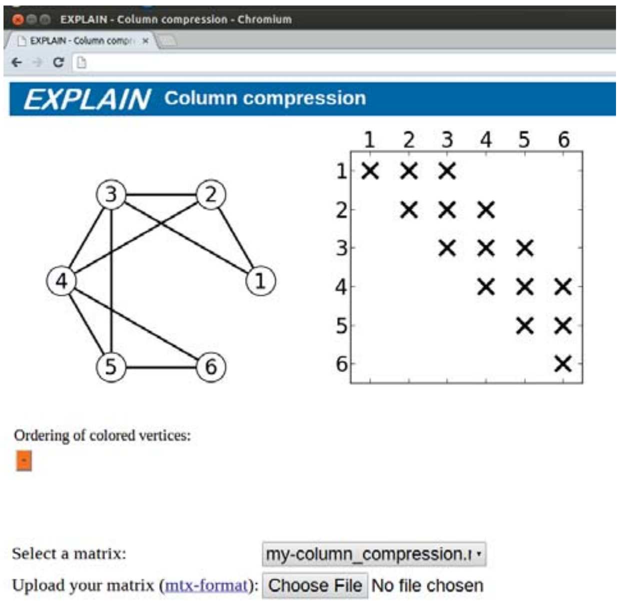

EXPLAIN - Column compression - Chromium
EXPLAIN Column compression
Ordering of colored vertices:
Select a matrix:
my-column_compression.
Upload your matrix (mtx-format):
Choose File
No file chosen

Figure 4.2: The initial layout of EXPLAIN for some given column compression problem.

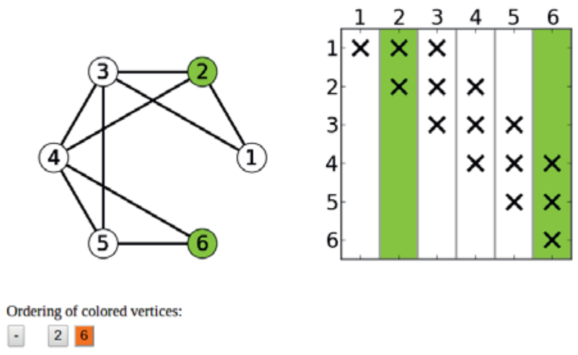
## 4.3 Available modules

In EXPLAIN, we have already implemented several modules. After discussing the module for the full unidirectional Jacobian compression using the column intersection graph, we explain the modules for the full and partial bidirectional Jacobian computation. Then, two other modules of nested dissection ordering and parallel matrix-vector product are presented to see other features of the software.

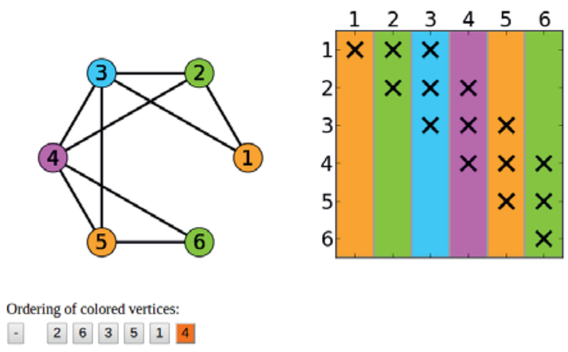
### 4.3.1 Column compression

In [16, 17], we presented a module of EXPLAIN which visualizes the coloring algorithm for the column compression interactively. Figure 4.2 shows a screenshot of the column compression module. The matrix and the corresponding column intersection graph are visualized beside each other. In the bottom of the page, different preloaded matrices can be selected or a new matrix can be uploaded from a file on the file system of the student's computer. The tool provides an interactive interface for the student who can control the algorithm such as returning to previous steps or loading different graphs and matrices. Selecting the vertices in different orderings generates different colorings corresponding to different column compressions.

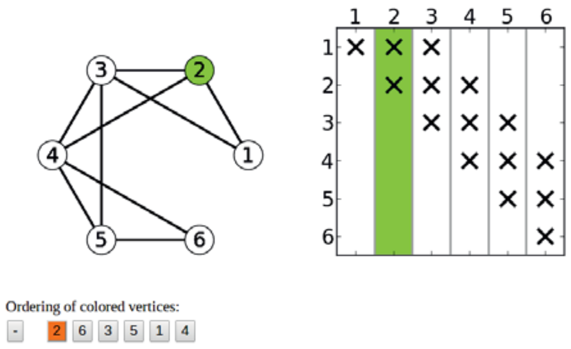
The module allows to select and, thus, color the vertices of a given graph step by step. The order in which the vertices are colored is interactively selected by the student. In each step, when the student selects a vertex, the program checks all of its neighbors regarding the colors. A color of the current step is then greedily selected from the predefined list of colors such that it differs from the colors of those neighbors that are already colored. To



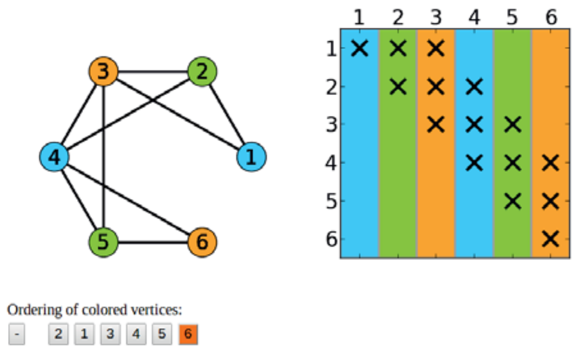
(a) Student selected $v_2$ and $v_6$ in that order.



(b) Student selected $v_2$, $v_6$, $v_3$, $v_5$, $v_1$, and $v_4$ in that order.



(c) Student selected vertices as in Figure 4.3(b) and then jumped back to $v_2$.



(d) Student selected $v_2$, $v_1$, $v_3$, $v_4$, $v_5$, and $v_6$ in that order.

Figure 4.3: Display of various situations after interactively choosing vertices.

indicate this, we do not color only the vertices in the graph but also the corresponding columns in the matrix.

Suppose a vertex is selected in the first step. This vertex is then colored using the first color of the predefined list. Continuing the process of vertex selection, different colors are chosen and an ordered list of vertices is created which is indicated in the subfigures of Figure 4.3 marked by "Ordering of colored vertices." Each button of this list is clickable, causing EXPLAIN to return to that step of the algorithm. The process continues until all vertices are colored. The button labeled by the minus sign will go back to the first step.

Figure 4.3(a) shows a representation of a nonzero pattern of the possible following matrix

$$J = \begin{bmatrix} 1 & 2 & 3 & 0 & 0 & 0 \\ 0 & 4 & 5 & 6 & 0 & 0 \\ 0 & 0 & 7 & 8 & 9 & 0 \\ 0 & 0 & 0 & 10 & 11 & 12 \\ 0 & 0 & 0 & 0 & 13 & 14 \\ 0 & 0 & 0 & 0 & 0 & 15 \end{bmatrix}, \tag{4.1}$$

and the related column intersection graph in which the student has already selected the two vertices $v_2$ and $v_6$. Both vertices are colored with the same color since they are not connected by an edge. Figure 4.3(b) represents the final step which shows that four colors are needed when the vertices are selected in the order $(v_2, v_6, v_3, v_5, v_1, v_4)$ displayed in the vertex list. The group of columns with the same color is compressed to a single column in the seed matrix as follows.

$$J \cdot S = J \cdot \begin{bmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 2 & 3 & 1 & 0 \\ 4 & 5 & 0 & 6 \\ 0 & 7 & 9 & 8 \\ 12 & 0 & 11 & 10 \\ 14 & 0 & 13 & 0 \\ 15 & 0 & 0 & 0 \end{bmatrix} . \tag{4.2}$$

Furthermore, the coloring of Figure 4.3(b) is the one corresponding to that compressed Jacobian (4.2).

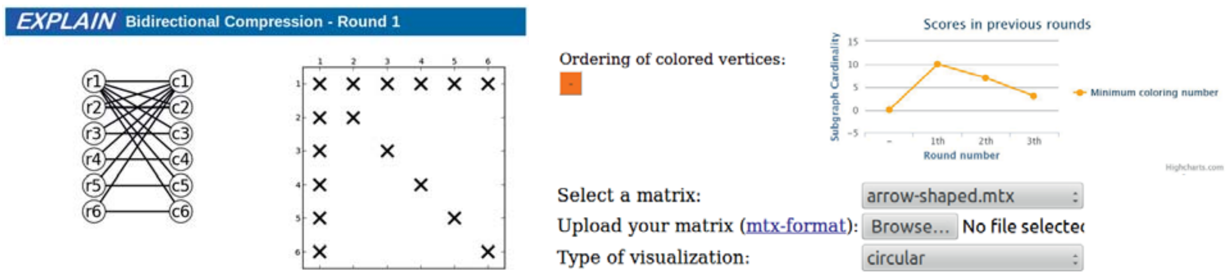
Since we want to provide the possibility to return to some step of the algorithm, a history of the selection process is kept in the ordered vertex list. Now, suppose the student selects to return to the step 1 where the vertex $v_2$ was selected, then the program returns to that step of the algorithm. The resulting state is depicted in Figure 4.3(c). Notice that the program keeps the whole history and the student can click on any other buttons in the history list.

On the other hand, the student can select a completely new selection order from the current step which can generate a smaller or larger number of colors. Employing the different ordering $(v_2, v_1, v_3, v_4, v_5, v_6)$ shown in Figure 4.3(d) leads to a reduction of one color compared to the first ordering given in Figure 4.3(b). In fact, this is the minimum number of colors needed to color this graph. The corresponding seed matrix is given by

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

## 4.3.2 Full and partial Jacobian computation

In our publication [19], we design and implement an interactive module to teach bidirectional compression and its connection to star bicoloring. Figure 4.4 shows an overview of the layout of the new module whose top and bottom part are shown in (a) and (b), respectively. In the top part, a graph and a matrix are visualized next to each other. Here, a matrix with a sparsity pattern in the form of an arrow is taken as an example. The nonzero pattern of the matrix is shown right and the corresponding bipartite graph is depicted left. A vertex $r_i$, which is placed on the left part of the graph, represents the $i$th row of the matrix. Likewise, a vertex on the right part of the graph labeled $c_i$ corresponds to the $i$th column of the matrix.



Ordering of colored vertices:

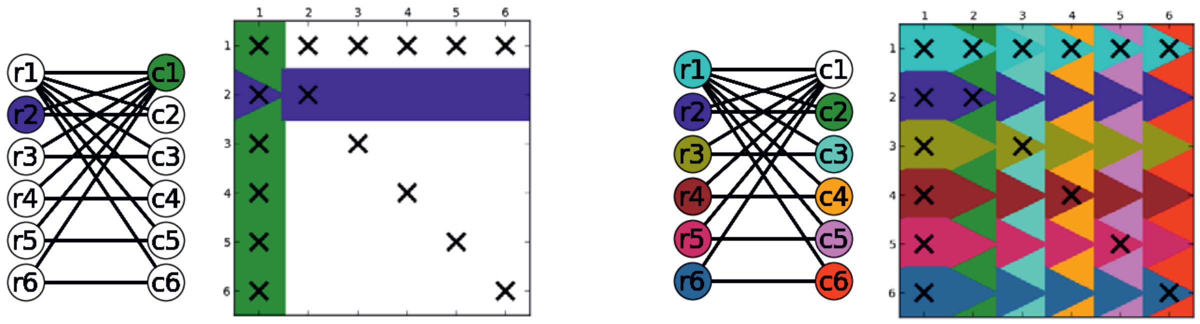
Select a matrix: arrow-shaped.mtx

Upload your matrix (mtx-format): No file selecte

Type of visualization: circular

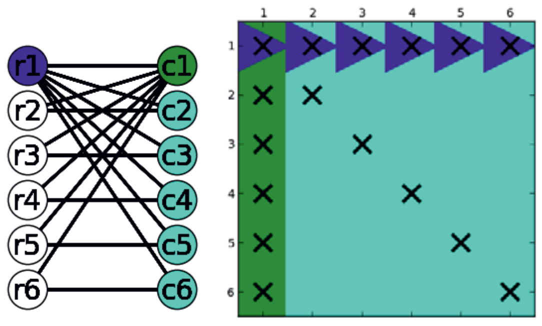
(a) (b)

Figure 4.4: The general layout of the bidirectional compression module. (a) The top part contains the visualization of the graph and its corresponding matrix. (b) The bottom part contains the intermediate steps, the input, and the history of selections.

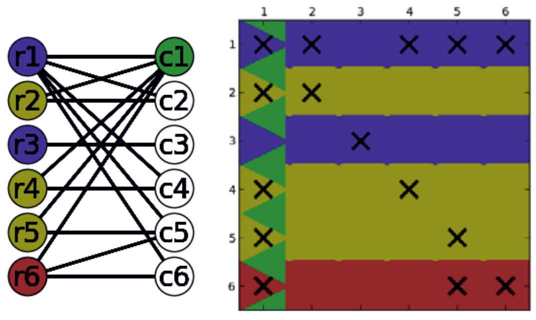


(a) (b)

Figure 4.5: The graph and the nonzero pattern (a) taken from Fig. 4.4 after the student interactively selected the vertices $r_2$ and $c_1$. A star bicoloring (b) of that example after trying to solve Minimum Star Bicoloring interactively. This star bicoloring uses 11 colors.



(a)



(b)

Figure 4.6: A star bicoloring (a) of the problem instance from Fig. 4.4 also considered in Fig. 4.5. This star bicoloring uses 3 colors and is an exact solution of Minimum Star Bicoloring. A star bicoloring (b) of a different problem instance using 4 colors which is also an exact solution of Minimum Star Bicoloring

Using any web browser, the student can interactively solve Problem 4, Minimum Star Bicoloring, by clicking on vertices of the bipartite graph. The selection of a vertex by a click refers to choosing this vertex to be colored next. This coloring is visualized simultaneously in the graph as well as in the matrix where the neutral color is the color white. By clicking on a row vertex, the vertex itself and the corresponding row is colored. This color should obey the rules specified in the definition of a star bicoloring. By clicking on a column vertex, this vertex and the corresponding column are colored. Recall that a nonzero element may be in both a colored column as well as in a colored row. In this case, we divide the square surrounding this element into a triangle and the remaining part. The triangle part is colored with the row color and the remaining part of the rectangle with the column color.

We now take the problem with the arrow-shaped nonzero pattern from Fig. 4.4 as an example. Here and in the following, we zoom into the graph and matrix view of the layout. The student interactively selects a sequence of row and column vertices to solve Minimum Star Bicoloring. Figure 4.5 (a) shows the situation after the student selected the vertices $r_2$ and $c_1$. The interactive selection then goes back and forth until a correct star bicoloring is found. Recall that the process of computing a solution of Minimum Star Bicoloring is called a round. The current round number is displayed at the top of the web page; see Fig. 4.4 (a). When a coloring is found at round number $x$, the page shows the message “Round $x$ is completed!”

Selecting vertices in different orders will typically result in different star bicolorings. A star bicoloring which is interactively chosen will not always have a minimal number of colors. For example, the order of vertex selection visualized in Fig. 4.5 (b) leads to a star bicoloring using 11 colors, which is obviously not the minimal number of colors. Here, all columns and rows are colored differently. In contrast, Fig. 4.6 (a) illustrates an exact solution of Minimum Star Bicoloring for this problem instance using the minimal

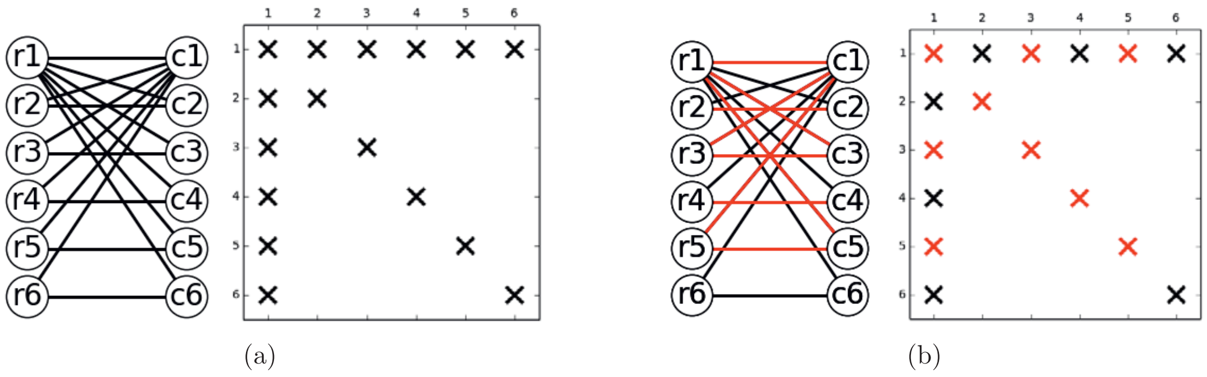


Figure 4.7: (a) The initial view of the module when no required edges are selected. (b) The user selects first the required edges. These required edges and the corresponding nonzero elements get the red color.

number of 3 colors.

After completing a round, the student can solve the same problem instance once more. In this case, the round number will be incremented, the colors will be removed, and another round is started using the initial situation depicted in Fig. 4.4 (a). The history of the number of non-neutral colors used in previous rounds is displayed below the matrix in a score diagram as shown in Fig 4.4 (b).

The subtle issues in understanding the connection between bidirectional compression and star bicoloring are more lucid when considering more irregularly-structured nonzero patterns. Another problem instance with a different nonzero pattern is shown in Fig. 4.6 (b). Here, it is more difficult to find out that this star bicoloring with 4 colors is indeed an exact solution to Minimum Star Bicoloring.

We extend this module to support the partial Jacobian computation. Here, the student should first select the required elements which are edges in bipartite graphs. So, when the student clicks on an edge, the color of this edge as well as the color of the corresponding nonzero element will be changed to red. These selected edges and nonzero elements are added to the required elements as shown in Figure 4.7.

As soon as the student starts to click the vertices, the required elements become fixed, i.e., no new required element can be added. The process of coloring is completely like the previous module. Figure 4.8 (a) shows a selection in which the student selects only column vertices. The result is a star bicoloring restricted to the red edges with 4 colors. A coloring with a smaller number of colors is shown in Figure 4.8 (b) in which a row vertex is selected first.

### 4.3.3 Nested dissection ordering

In our paper [18], we present the nested dissection module to illustrate the connection between the following scientific computing and combinatorial problems,

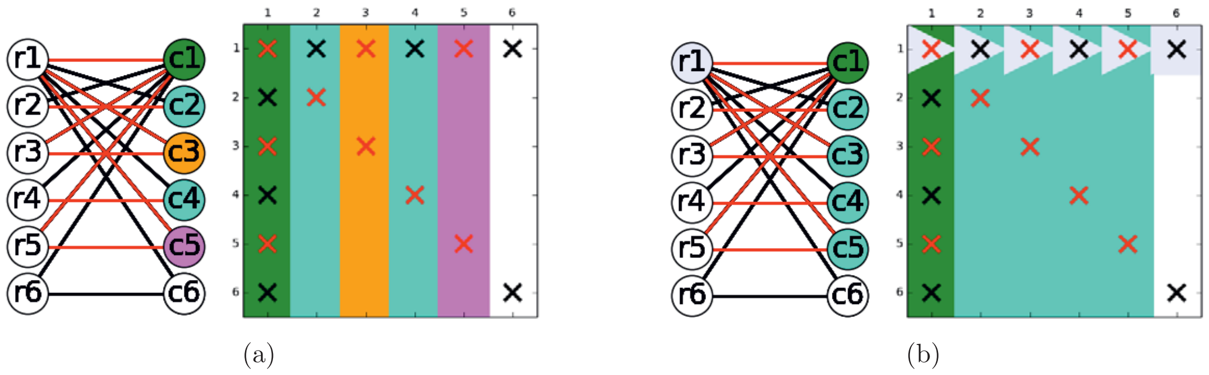


Figure 4.8: A star bicoloring restricted to the red edges (a) with four colors in which only column vertices are selected and (b) with three colors in which the user colors a row vertex first.

**Problem 11 (Nested Dissection Ordering)** *Given a sparse symmetric positive definite matrix $A$, find a symmetric permutation $P^TAP$ of $A$ in the form of*

$$A' = \begin{bmatrix} A_1 & 0 & B_1^T \\ 0 & A_2 & B_2^T \\ B_1 & B_2 & C \end{bmatrix}, \tag{4.3}$$

*such that the size of the block $C$ is minimized while the sizes of the blocks $A_1$ and $A_2$ are balanced.*

**Problem 12 (Small Vertex Separator)** *Given the graph $G$ associated with a sparse matrix $A$, find a disjoint decomposition of the vertices $V = V_1 \cup V_2 \cup S$ with a vertex separator $S$ such that the size of the vertex separator, $|S|$, is minimized while the sizes of the two remaining components, $|V_1|$ and $|V_2|$, are balanced.*

The graph model of this problem has the simple graph layout considering the matrix as the adjacency matrix of a graph. The algorithm from [18] searches for a vertex-separator set corresponding to the block $C$. Here, a vertex separator $S$ of the given graph $G$ is a subgraph of $G$ if the removal of $S$ and its adjacent edges from $G$ results in two disconnected components $V_1$ and $V_2$ of $G$. Suppose we move rows and columns corresponding to the vertex separator to the end of the matrix and bring together the corresponding columns of $V_1$ and $V_2$ by a permutation, then a nested dissection is visualized in the matrix by the block form (4.3).

In this module representing a bisection, a round consists of finding a vertex separator. Figure 4.9 shows the initial layout of an example and the selection of the vertex $v_{10}$ for the set of vertex separator. The module moves the column corresponding to the selected vertex to the end of the matrix and both the vertex and its corresponding column get the orange color. Additionally, the adjacent edges of the vertex are removed for a better visualization (compare Figure 4.9 (a) and Figure 4.9 (b)).

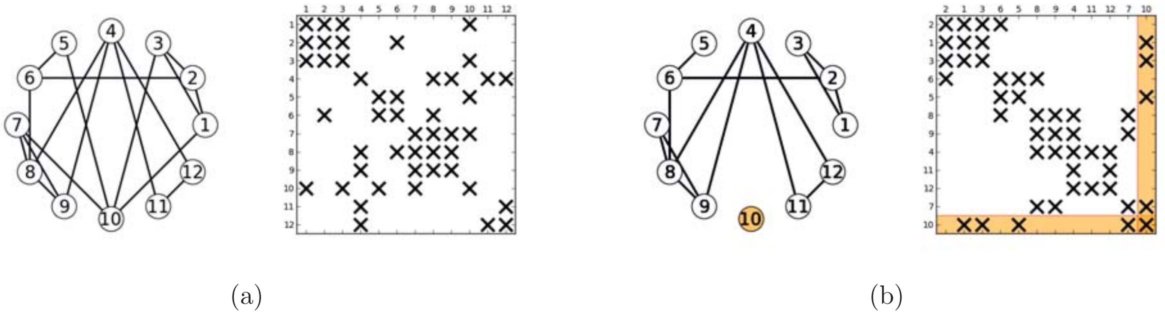


Figure 4.9: (a) Two equivalent representations in terms of a graph and a matrix. (b) Graph and matrix view after selecting the vertex number 10. The decomposition into two blocks is still not shown as the graph is not yet decomposed into two disconnected components.

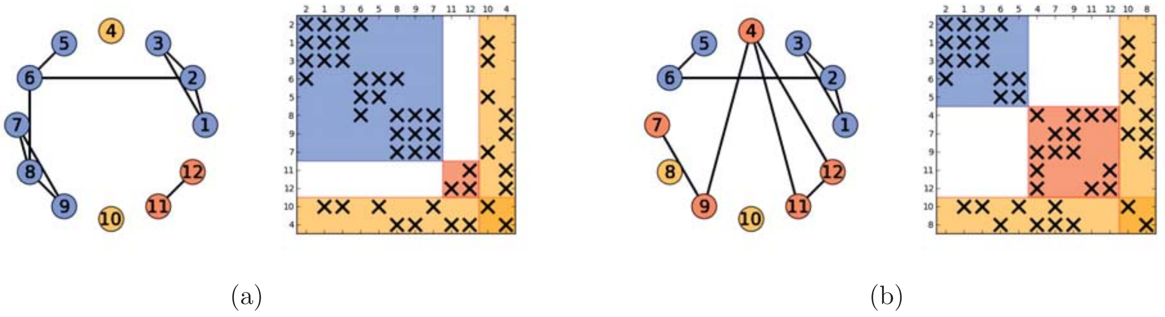


Figure 4.10: (a) Graph and matrix view after selecting the vertices number 10 and then 4. The selection is not adequate as the sizes of blocks are not balanced. (b) Graph and matrix view after selecting the vertices number 10 and then 8. The block sizes are balanced and the separator size is minimized.

Figure 4.10 shows two scenarios. We have an unbalanced result in Figure 4.10(a) in which the vertex $v_4$ is selected after the initial selection of $v_{10}$. However, Figure 4.10(b) results in a balanced result by only selecting $v_8$ instead of $v_4$. When a vertex separator is found, the software performs the permutation and colors the two disconnected components of the graph as well as the distinct blocks on the diagonal with blue and red.

The feedback diagram from different rounds looks like Figure 4.11. Here, the blue and red curves should have values close to each other since this indicates the balancing condition. The orange curve should be as small as possible because it represents the minimization of the separator size. In the last round of this figure, we see a balanced result with minimal separator size.

Based on the new features of EXPLAIN 2.0, we update the previous bisection to a recursive bisection algorithm. It contains the bisection itself. So, the student selects the

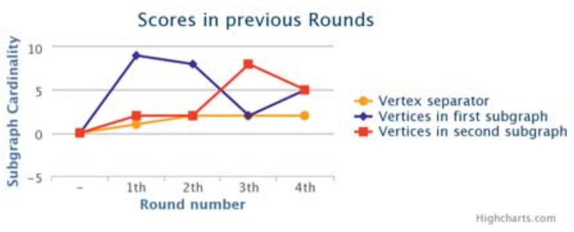


Figure 4.11: Score diagram resulting from four different rounds.

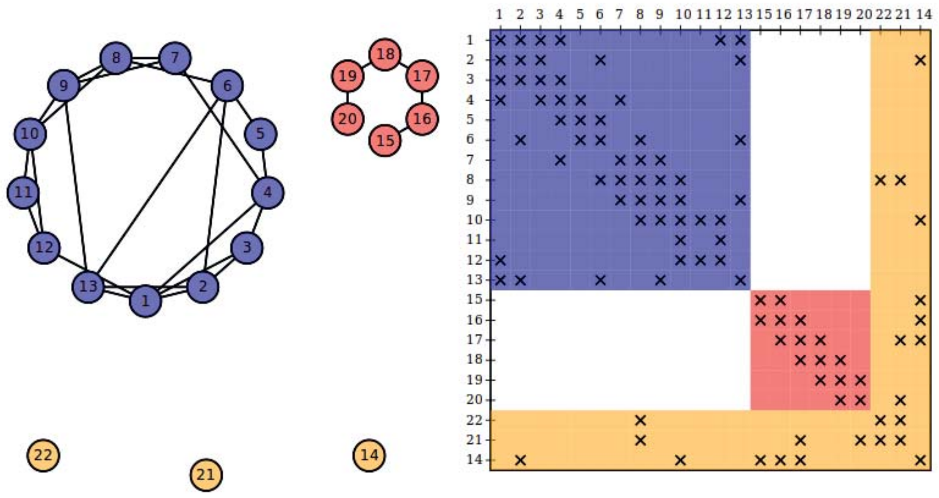


Figure 4.12: A selection results in an unbalanced bisection of graph.

vertex separator as before until the graph becomes disconnected. In this new version, the vertex separator is shown in orange on the bottom of the graph. Also the two remaining components of the graph are shown separately in two different colored circles at the top of this figure. In Figure 4.12, the result of a selection is visualized which is not balanced. Figure 4.13 also shows the results of a more balanced selection.

Now, in contrast to the previous version, the student can click further on the vertices of each component. This selection would trigger a recursive bisection of the two components of the matrix as well. Again, this selection goes forward until both graph components become disconnected. The vertex separators are moved to the bottom of the graph components as well as the graph components are shown separately. Figure 4.14 and Figure 4.15 illustrate an unbalanced result and a better balanced result of nested dissection, respectively. Figure 4.16 shows how the results can get even more balanced.

The corresponding feedback diagram is modified such that both the size of the vertex separator as well as the size of the four graph components can be visualized. In this diagram, the separator size shows the sum of all the vertex separators of all recursion levels. Figure 4.17 shows a possible selection history. The line chart shows the size history of the vertex separator and four-bar chart grouped together shows the size history of the graph

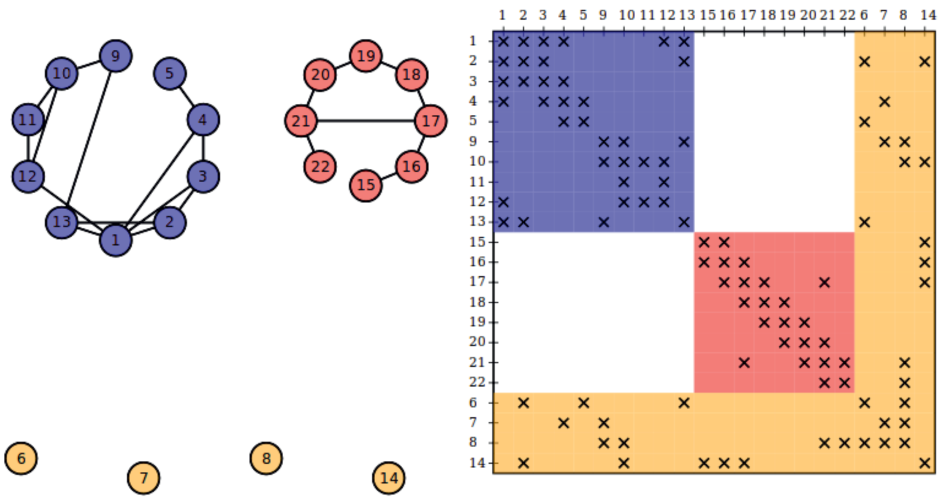


Figure 4.13: A selection results in a more balanced bisection of graph.

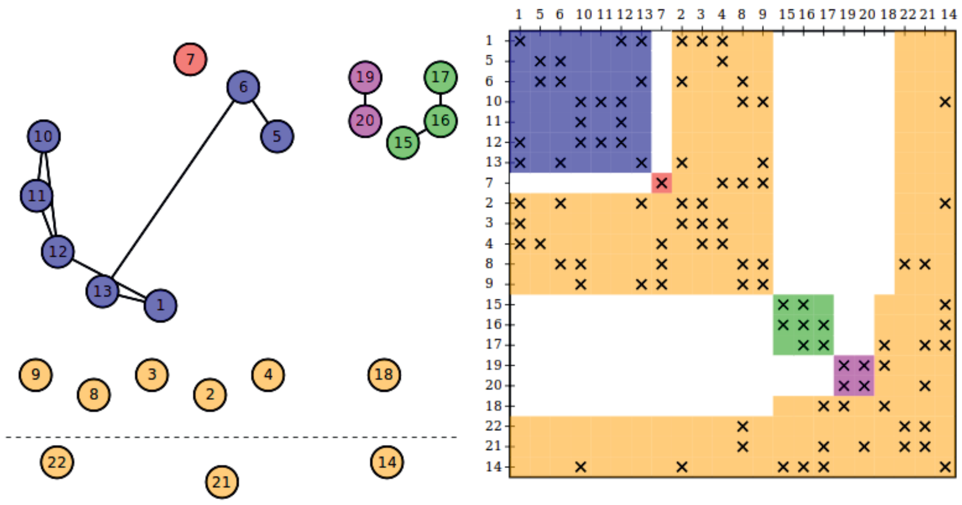


Figure 4.14: A complete nested dissection with an unbalanced result.

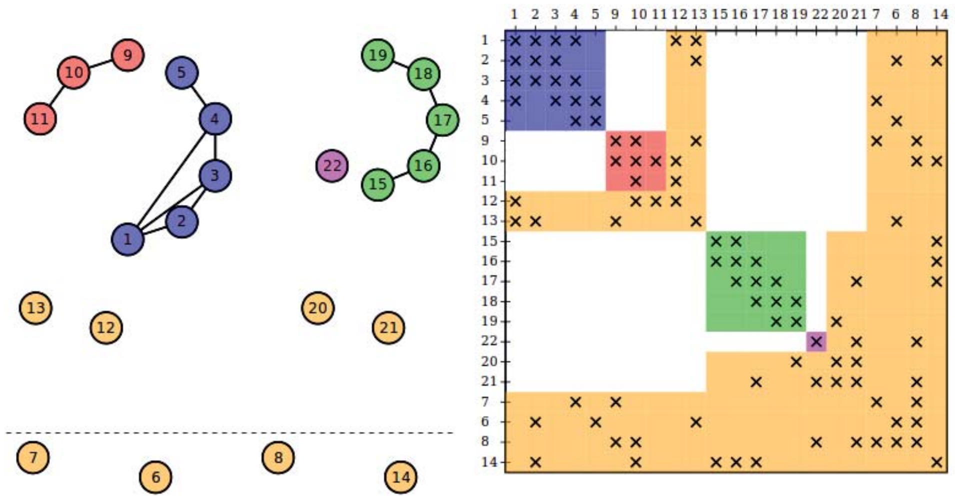


Figure 4.15: A complete nested dissection with a better balanced result.

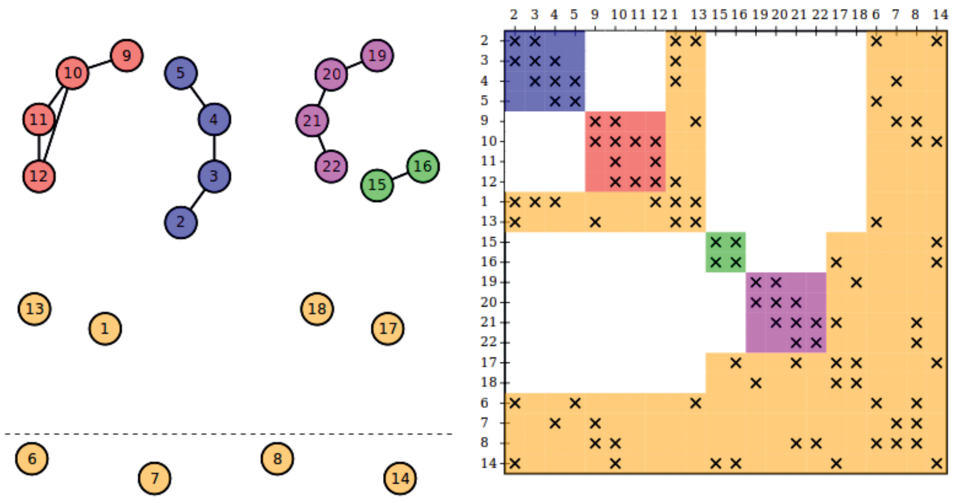


Figure 4.16: A complete nested dissection with an even more balanced result.

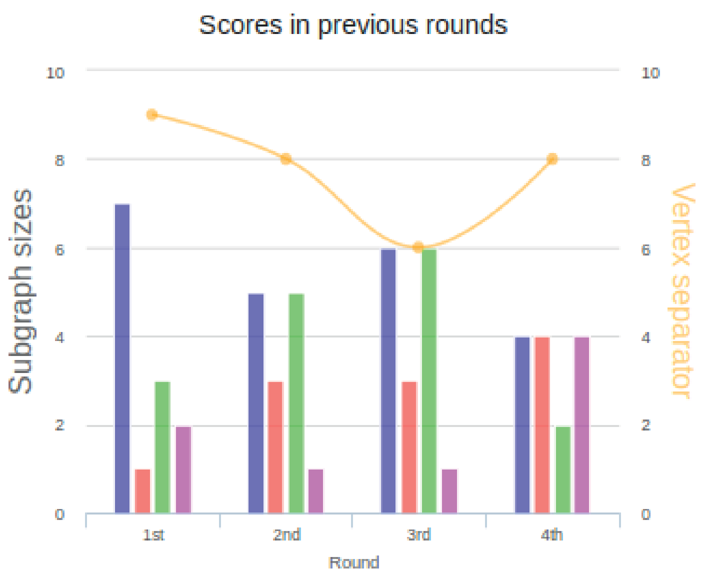


Figure 4.17: The size of four subgraphs, shown as bar charts, resulted from nested dissection compared in different rounds. The colors of the bars are related to the colors of the corresponding subgraphs. The curve in orange color shows the size of the vertex separator.

parts. The colors are also the same colors used in the graph and matrix visualizations. The goal here is to minimize the size of the vertex separator as well as balancing the size of the subgraphs. We achieved a balanced results in the fourth round of selection.

### 4.3.4 Parallel matrix-vector product

In our paper [20], we present a module for a parallel matrix-vector product to illustrate the connection between the following scientific computing and combinatorial problems.

**Problem 13 (Data Distribution)** *Given a large sparse matrix $A$ with a symmetric nonzero pattern and a dense vector $x$. Suppose we want to compute the sparse matrix-vector product $y \leftarrow Ax$ on a computer with distributed memory. Find a non-redundant data distribution of the nonzero elements of $A$ in a row-wise fashion and a consistent distribution of $x$ and $y$ such that the communication between processes is minimized while the number of arithmetic operations is balanced between the processes.*

**Problem 14 (Graph Partitioning)** *Given an undirected connected graph $G$, find a partition of the vertices $V$ into nonempty disjoint subsets such that the number of edges with incident vertices in different partitions is minimized while the number of vertices of the subsets is balanced.*

These two problems are connected by letting a row $i$ of $A$ be represented by a vertex $v_i$ in $G$ and the nonzero elements in the positions $(i, j)$ and $(j, i)$ correspond to an edge between vertices $v_i$ and $v_j$. Let $P : V \to \{1, 2, \ldots, p\}$ be the vertex partition to $p$ processes that decomposes the set of nodes $V$ of the graph into $p$ subsets $V_1$, $V_2$, $\ldots$, $V_p$ such that

$$V = V_1 \cup V_2 \cup \cdots \cup V_p$$

with $V_i \cap V_j = \emptyset$ for $i \neq j$. This decomposition of $V$ represents the distribution of the rows of $A$ to $p$ processes.

Assuming that the number of nonzeros is roughly the same for each row of $A$, the computation is evenly balanced among the $p$ processes if the partition $P$ is $\varepsilon$-balanced defined as

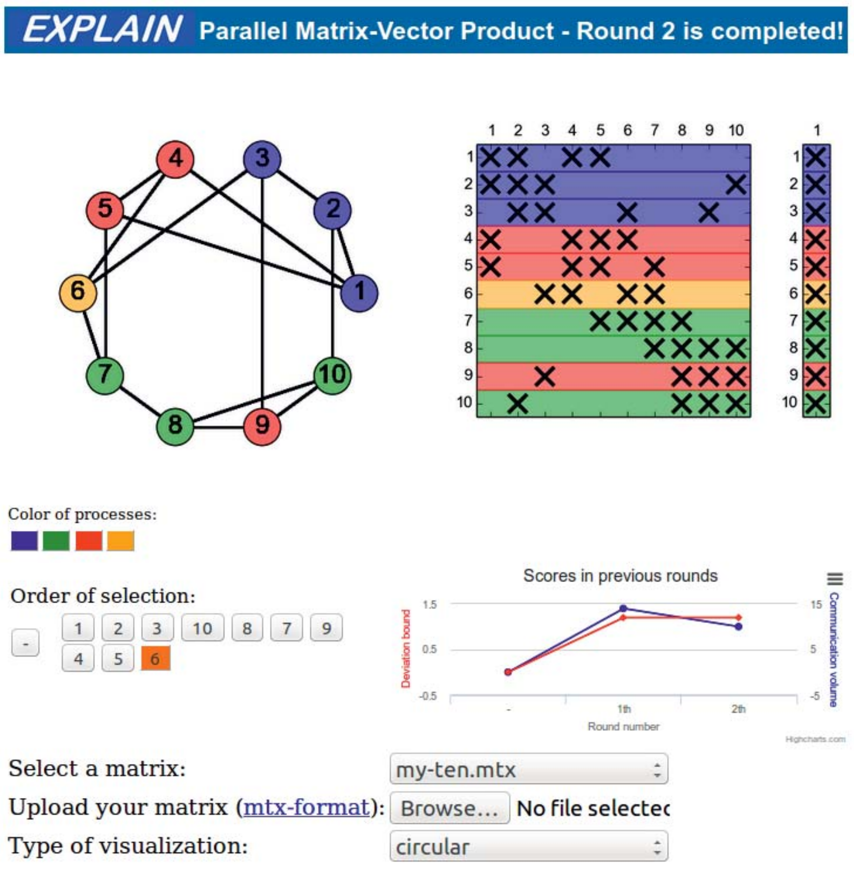
$$\max_{1 \leq i \leq p} |V_i| \leq (1 + \varepsilon) \frac{|V|}{p}, \tag{4.4}$$

for some given $\varepsilon > 0$. The graph partitioning problem consists of minimizing the number of edges with incident vertices in different partitions (the cut size) of an $\varepsilon$-balanced partition. It is a hard combinatorial problem [59].

The parameter $\varepsilon$ introduced in (4.4) is used to quantify the degree of imbalance allowed in a data distribution. If $\varepsilon = 0$ all processes are assigned exactly $|V|/p$ rows of $A$, meaning that no imbalance is allowed at all. When increasing $\varepsilon$ the load balancing condition (4.4) is relaxed. The larger $\varepsilon$ is chosen, the larger is the allowed imbalance. Thus, in some way, $\varepsilon$ quantifies the deviation from a perfect load balance. An equivalent form of (4.4) is given by

$$\frac{p}{|V|} \max_{1 \leq i \leq p} |V_i| - 1 \leq \varepsilon, \tag{4.5}$$

which can be interpreted as follows. Suppose that you are not looking for an $\varepsilon$-balanced partition $P$ for a given $\varepsilon$, but rather turn this procedure around and ask: "Given a partition



Select a matrix: my-ten.mtx

Upload your matrix (mtx-format): No file selectec

Type of visualization: circular

Figure 4.18: Overall structure of the sparse matrix-vector multiplication module.

$P$, how large need $\varepsilon$ at least be so that this partition is $\varepsilon$-balanced?" Then the left-hand side of the inequality (4.5) which we call *deviation bound* gives an answer to that question. The extreme cases for the deviation bound are given by 0 if the distribution is perfectly balanced and $p-1$ if there is one process that gets all the data.

Figure 4.18 shows the overall layout of this interactive module for sparse matrix-vector multiplication. The top of this figure visualizes the representation of the problem regarding the graph $G$ as well as in terms of the matrix $A$ and the vector $x$. Below on the left, there is a panel of colors representing different processes and another panel displaying the order of selecting vertices of the graph. Next, on the right, there is a feedback diagram recording values characterizing communication and load balancing.

This figure gives an overall impression of the status of the module after a data distribution is completed. Here, $p = 4$ processes represented by the colors blue, green, red, and yellow get data by interactive actions taken by the student. Figure 4.19 now shows the status of the module in a phase that is more related to the beginning of that interactive procedure. For a given matrix, the student can distribute the data to the processes by first clicking on a color and then clicking on an arbitrary number of vertices. That is, the distribution of vertices to a single process is determined by first clicking on a color $j$ and then clicking

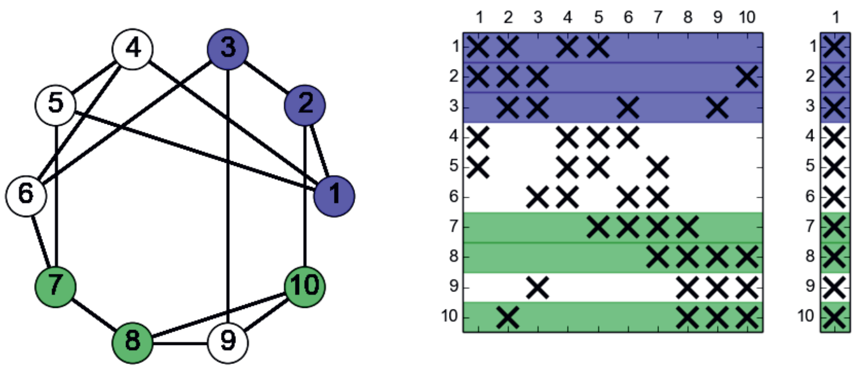


Figure 4.19: The intermediate state after the student selected six vertices.

on a particular number of vertices such that these vertices are distributed to the process $j$. Then, by clicking on the next color, this procedure can be repeated until all vertices are interactively colored and, thus, the data distribution $P$ is finally determined.

Figure 4.19 illustrates the situation after the student distributed vertices $\{v_1, v_2, v_3\}$ to the blue process and the vertices $\{v_7, v_8, v_{10}\}$ to the green process. By interactively assigning a vertex to a process, not only the vertex is colored by the color representing this process, but also the row in the matrix as well as the corresponding vector entry of $x$ are simultaneously colored with the same color. This way, the data distribution is visualized in the graph and the matrix simultaneously which emphasizes the connection between the matrix representation and the graph representation of that problem. By inspection of the panel representing the order of selection in Figure 4.18, we find out that the status depicted in Figure 4.19 is an intermediate step of the interactive session that led to the data distribution in Figure 4.18. Any box labeled with the number of the chosen vertex in that panel is also clickable allowing the student to return to any intermediate state and start a rearrangement of the data distribution from that state.

In this module, the problem consists of distributing all data needed to compute the matrix-vector product to the processes. Equivalently, the distribution of all vertices of the corresponding graph to the processes is a round. Suppose that round 2 is completed as given in Figure 4.18. Then, the student can explore the data distribution in more detail by clicking on a color in the panel labeled “Color of processes.” Suppose that the student chooses the red process, then this action will modify the appearance of the vector $x$ in the matrix representation to the state given in Figure 4.20. Here, all vector entries that need to be communicated to the red process are now also colored red. The background color in the vector still represents the process that stores that vector entry.

After completion of a round, it is also instructive to focus on the quality of the data distribution $P$. Recall that the graph partitioning problem aims at minimizing the cut size of $P$ while balancing the computational load evenly among the processes. Figure 4.21 shows the feedback diagram. For each round, this diagram shows the cut size using the label “communication volume.” In that feedback diagram, the student can attempt to minimize the communication volume over some rounds.

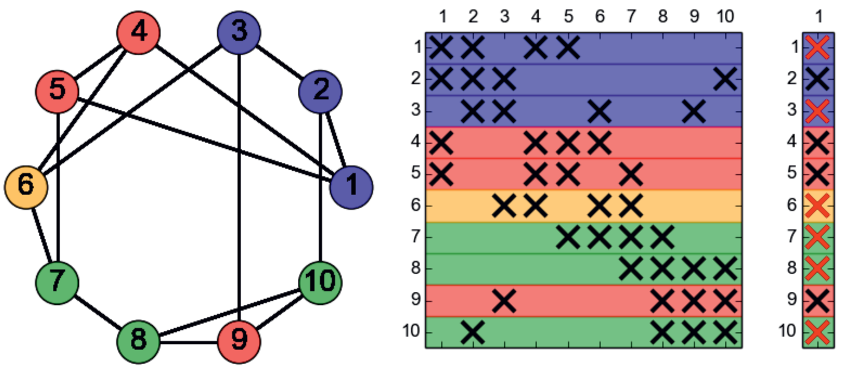


Figure 4.20: All vector entries $x_i$ to be communicated to the red process are drawn in red.

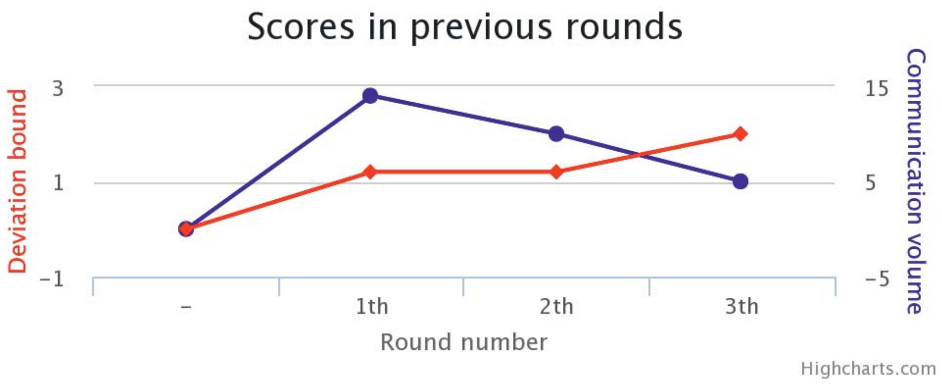


Figure 4.21: The communication volume and the deviation bound versus various rounds.

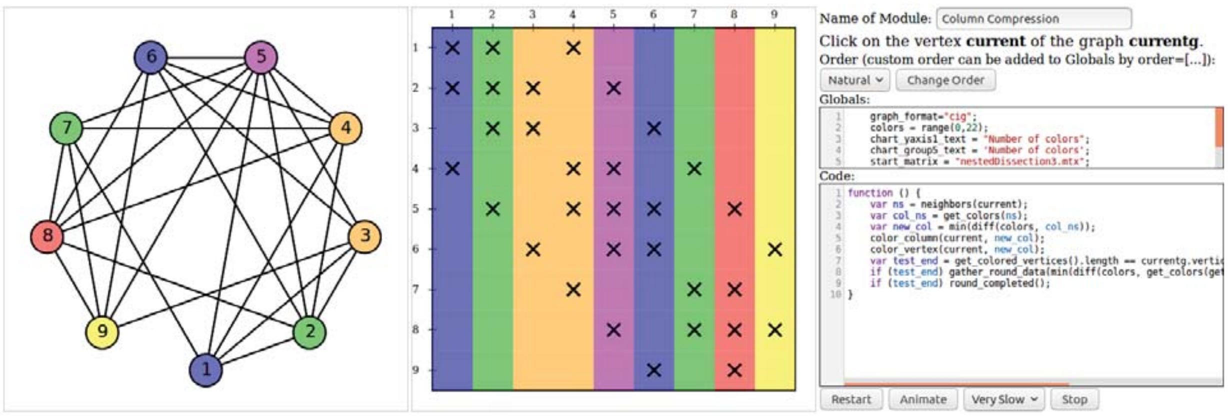
The feedback diagram also shows the value of the deviation bound for each round. A low deviation bound indicates a partition that balances the computational load evenly, whereas a large deviation bound represents a significant imbalance of the load. The score diagram helps the student to evaluate the quality of a single data distribution and to compare it with distributions obtained in previous rounds. This feedback to the student is in the spirit of computer games, where a score has only a low immediate relevance to the current game. However, the idea is to achieve a "high score" and try to motivate the player to beat that score in subsequent rounds, thus offering an extra challenge. For this educational module, a "high score" would consist of a low communication value together with a small deviation bound.

## 4.4 New features in EXPLAIN 2.0

EXPLAIN 2.0 has various features which are discussed in the following paragraphs. The new major feature is an algorithm editor. In EXPLAIN 2.0, the student can see and edit the source code of an algorithm beside the visualization of the graph and matrix. There is a new button in the control button named as *Edit Algorithm!*. Clicking this button shows an editor with the source code of the corresponding module. This source code is written in a simple scripting language. Figure 4.22 illustrates the column compression module. As it can be seen, an editor with the corresponding source code of the column compression module is shown in the right part of the figure. Additionally, the student can see an animation of this algorithm by first selecting and ordering and then clicking the button named as *animate*. An animation goes through the vertices and executes each line of the algorithm one by one to visualize the algorithm when it is executed. The student can stop the algorithm and select the speed of execution.

Figure 4.23 shows the control panel of EXPLAIN 2.0 for the nested dissection module in the left figure and the column compression module in the right figure. Compared to EXPLAIN 1.0, we here have three new buttons for the three functionalities: going to next round, doing a postprocessing, and editing an algorithm. The first button named as *Go to next round* goes to the next round even if the current round is not completed. The second button is for a postprocessing step. A postprocessing is an action which can be done when a round is completed. It can be different for each module. For example, Figure 4.23 (Left) has the postprocessing named as *Show edges*. This shows the edges between the vertex separators and the subgraphs which we remove during the selection. Another example is Figure 4.23 (Right) which does not have any postprocessing. So, the button is disabled. The third button is to show and hide the algorithm editor. The label of the button is first *Edit Algorithm!* and changed to *Finish Editing!* after clicking.

Another minor change is the reference to a publication which explains the module. Figure 4.23 (Left) and (Right) shows the two publications [16] and [18] for the nested dissection and the column compression module and the link to the publishers.



Name of Module: Column Compression

Click on the vertex **current** of the graph **currentg**.

Order (custom order can be added to Globals by order=[...]):

Natural | Change Order

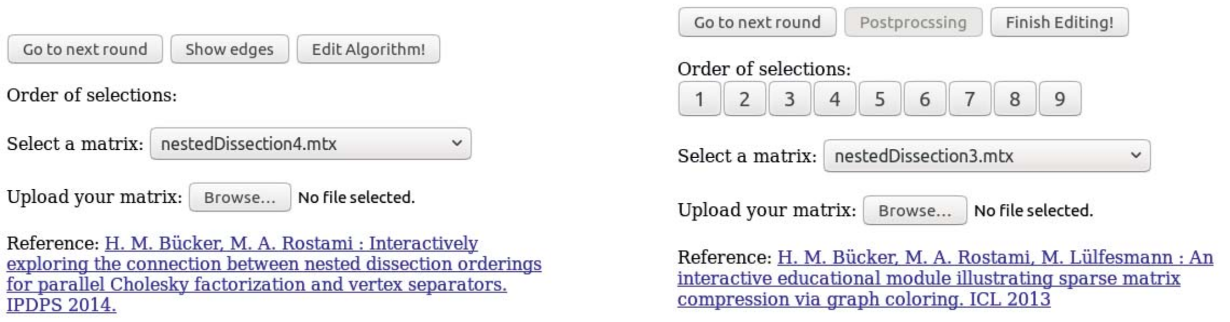
Globals:

```
    graph_format="cig";
    colors = range(0,22);
    chart_yaxis1_text = "Number of colors";
    chart_group5_text = 'Number of colors';
    start_matrix = "nestedDissection3.mtx";
```

Code:

```
function () {
    var ns = neighbors(current);
    var col_ns = get_colors(ns);
    var new_col = min(diff(colors, col_ns));
    color_column(current, new_col);
    color_vertex(current, new_col);
    var test_end = get_colored_vertices().length == currentg.vertic
    if (test_end) gather_round_data(min(diff(colors, get_colors(get
    if (test_end) round_completed();
}
```

Figure 4.22: The code of the corresponding module is visualized beside graph and matrix. This code is in a simple scripting language. The user specifies the order and this code is executed based on that order.



Order of selections:

Select a matrix: nestedDissection4.mtx

Upload your matrix: No file selected.

Reference: H. M. Bücker, M. A. Rostami : Interactively exploring the connection between nested dissection orderings for parallel Cholesky factorization and vertex separators. IPDPS 2014.

Order of selections:

Select a matrix: nestedDissection3.mtx

Upload your matrix: No file selected.

Reference: H. M. Bücker, M. A. Rostami, M. Lülfesmann : An interactive educational module illustrating sparse matrix compression via graph coloring. ICL 2013

Figure 4.23: The control panel of EXPLAIN 2.0 is visualized. It has three new buttons for going to the next round, doing a postprocessing step, and editing the algorithm. Additionally, it has a new reference link to a publication for this module. (Left) The control panel is visualized for the nested dissection module. Here, the postprocessing step is to show the missing edges which we remove during the selection processes. (Right) The control panel is visualized for the column compression module. In this module, we do not have any postprocessing step. Hence, the button is disabled.

## 4.5 Implementation details of EXPLAIN

Lülfesmann et al. [60] first introduced an standalone version of EXPLAIN that needed a client with administrator privileges to install Python libraries as well as the software itself. Here, we introduce two new releases of EXPLAIN.

### 4.5.1 Version 1.0

This section is a summary of the implementation details of EXPLAIN 1.0 in our publication [16]. In EXPLAIN 1.0, the software is moved to the online platform which means the student needs just a web browser to work with the software. EXPLAIN 1.0 combines several Python packages. More precisely, the graph data structure is handled by *NetworkX* [61]. It provides different operations like creation and deletion of vertices and edges. It also allows the programmer to access characteristic graph information such as the neighbor vertices and the number of vertices. Using this library together with *matplotlib* [62] covers the different aspects of visualization. This library produces different layouts of graphs as well as the properties of vertices and edges. The matrix manipulation and visualization are handled by *Scipy* [63], specifically the construction and the visual layout of sparse matrices.

The conversion from the standalone to the online version needs the Python library *Mod_python* [64]. It comes from the *Apache* project including the Python interpreter in the given Apache web server. Using this library helps to keep the previous program structure as much as possible.

The library *Mod_python* assists to implement folder management, user interaction, cookie handling, and the web interface. Specifically, the *Mod_python* modules like *Apache*, *util*, and *PSession* are used to migrate the previous version of EXPLAIN to a web version. As already mentioned, the Python interpreter is embedded into the web server by the *Mod_python* module.

In the standalone version, an event was handled by a local Python function but, in the new version, there are two sides: server and client. The web browser at the client side shows HTML websites with embedded Javascript source code while the server side is a Python server. Since the buttons are HTML buttons and the events are Javascript functions, a Javascript function submits a form to the server containing the execution request and parameters to the related Python function. Then, the called Python function writes the dynamically generated result as an HTML string to the Apache request. The server sends back the HTML string and the client shows the string as a web page.

As an example, the basic algorithm of coloring in the column compression module and keeping the history is shown in the pseudo-codes given in Figure 4.24. The first procedure represents what happens when a student clicks on a vertex. The second one shows how the history of matrix and graph images are loaded when the student clicks on one of the history buttons.

The first procedure, VertexClicked, takes the selected vertex $v$ as an input parameter. To color this vertex $v$, it finds the first color from the list *ColorList* that is different from the colors of the neighbors of $v$. The coloring of the graph is then changed, shown, and saved as an image. Also, the vertex $v$ is added to the ordered list, $Hist$, of selected vertices for the history.

The second procedure, HistClicked, takes the selected history $h$. This history will be used to find and plot the previously stored images of the matrix and the graph. Also, the variable $IsInHist$ specifies that the program is in the "history mode" which is important if the user selects a vertex different from the previous order. In this case, the program overwrites the current history and saves new images.

### 4.5.2 Version 2.0

In EXPLAIN 2.0, we changed the code such that it becomes more efficient and easier to extend. In the previous version, the module was mainly based on the Python libraries: *NetworkX* [61] for the graph data structure, *matplotlib* [62] for the visualization aspects, *Scipy* [63] for the sparse matrix computation, and *Mod_python* [64] for the web-based version.

There were two problems with the previous implementation. First, the final visualization of graph and matrix was always an image. So, the time for saving and loading the image was always a problem. Second, since the final result was HTML/JS code and the computation part was in Python, an overhead of the server management for *Mod_python* is always added to the system.

In the new implementation, we replace all Python parts with the Javascript code. We choose the Javascript library D3 (Data-Driven Documents) because of its power of control and visualization. Figure 4.25 shows an illustration of the data structure of adjacency list for graph which is implemented in Javascript. There is an the array representing the vertices. Each cell of this array points to another array *edges* which contains the identity of the vertices which are neighbors of that vertex. Here, we show that the data structure contains other properties like colors. Using the object structure of Javascript, it can be extended dynamically to include any other properties which may be necessary later. For example, two properties *distance* and *parent* are added in the implementation of Breadth-First Search (BFS).

We consider a model-view-control (MVC) design pattern [65] for our implementation. An important aspect of our design is the suitable connection of the model and view. A practical approach enables the direct change in view while it keeps the separation of the view and model. So, we have unique ids for edges and vertices. The unique ids for vertices are the concatenation of the string "ver" and the actual id of the vertex. Similarly, the unique ids for edges are defined as the concatenation of the four strings "edge", the source vertex, "-", and the target vertex. The same idea applies to the matrix view. Each cell of the matrix is accessible through a unique id combining the strings "cell", the row index, "-", and the column index. Each nonzero of the matrix also gets the unique id that is built in a similar way as the cell id, but replacing the string "cell" by the string "nnz".

The previous discussion of the view access enables us to select the specific element and change its behavior and properties. For example, the following code changes the color of

```
$ColorList \leftarrow \{\text{green, turquoise, orange, violet, ...}\}$
$Hist \leftarrow \{\}$                                          ▷ History of selected vertices.
$WhereInHist \leftarrow 0$                                      ▷ Where in history are we?
$IsInHist \leftarrow False$                                     ▷ Are we in history mode?

procedure VertexClicked($v$)                                    ▷ User clicks vertex $v$.
    $ns \leftarrow$ neighbors($v$)
    $ColorIndex \leftarrow 1$                                   ▷ Allowed color index
    for $i \leftarrow 1$ to size($ColorList$) do
        $AllowedColor \leftarrow True$
        for $j \leftarrow 1$ to size($ns$) do
            if $ColorList[i] =$ color($ns[j]$) then
                $AllowedColor \leftarrow False$
        if $AllowedColor = True$ then
            $ColorIndex \leftarrow i$
            Break
    Color $v$ with the color $ColorList[ColorIndex]$

    if graph and matrix images are not already saved then
        SaveMatrix()                                            ▷ Using a specific name
        SaveGraph()                                             ▷ Using a specific name
    if $IsInHist = True$ then
        for $i \leftarrow WhereInHist + 1$ to size($Hist$) do
            $Hist.removeElementAtPosition(i)$
        $Hist$.add($v$)
        $IsInHist \leftarrow False$
    else
        $Hist$.add($v$)
    Update($Hist$)                                              ▷ Update history buttons


procedure HistClicked($h$)                                      ▷ User clicks history $h$.
    OpenMatrix($h$)                                             ▷ Plot/load matrix with specific name
    OpenGraph($h$)                                              ▷ Plot/load graph with specific name
    $WhereInHist \leftarrow find(Hist, h)$                      ▷ The location of $h$
    $IsInHist \leftarrow True$
```

Figure 4.24: Pseudocode of the event handling in EXPLAIN

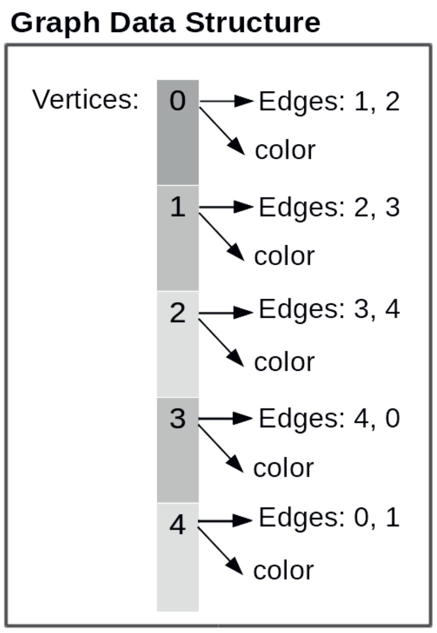


Figure 4.25: The graph data structure which is implemented in EXPLAIN 2.0.

the vertex with the id *i* to the red color and the color of a matrix cell to the blue color,

```
d3.select("#ver"+i).set_color(red);
d3.select("#cell"+i+"-"+j).set_color(blue);
```

Another aspect of the implementation is the order of drawing edges and vertices. Since we do not want to draw the edges on the top of the vertices, the edges should be drawn first. On the other hand, we draw an edge only by getting the positions of its vertices. This direct connection helps to avoid the several updates of view for drawing vertices and edges. To solve this problem, we draw the edges and vertices in order by using the grouping concepts of D3.

After the first design of the software, we then considered the actual interface for the developer. Since we do not need all the functionality which the programming language provides, we design a new scripting language which has particular functions for working with matrix and graph. Table 4.1 shows some of these functions.

Having such scripting language empowers us to have a dynamic scripting editor together with EXPLAIN which makes the creation of new modules efficient and fast. The developer of a new module needs only to write the action command of the vertex click and the global variables which he/she needs. There are some predefined variables for required data. As an example, the variable *current* and *colors* represents the current vertex and the list of predefined colors, respectively. The following code shows the code for the column compression module as an example.

```
var ns = neighbors(current);
var col_ns = get_colors(ns);
```

| | |
|---|---|
| neighbors($l_v$) | returns the neighbors of the list $l_v$ of vertices |
| color_vertex($v$,$c$) | colors the vertex $v$ with the color $c$ |
| color_column($i$,$c$) | colors the column $i$ with the color $c$ |
| color_row($j$,$c$) | colors the row $j$ with the color $c$ |
| min($l$) and max($l$) | finds minimum and maximum of the list of integers $l$ |
| diff($A$,$B$) | finds $A - B$ |
| get_colors($l_v$) | returns a list of colors of the list $l_v$ of vertices |

Table 4.1: A list of functions available in the new scripting language.

```
var new_col = min(diff(colors,col_ns));
color_column(current, new_col);
color_vertex(current,new_col);
```

Each module in EXPLAIN 2.0 consists of four functions with particular name conventions. For example, the following list represents these functions for the column compression module. Here, all functions except the one which computes one step of the algorithm have the ending made up from the first characters of the name of the module.

- column_compression: The function which computes one step of the algorithm.
- reference_cc: This function defines two strings: a bibliography for a reference (*reference_text*) and the actual reference url (*reference_url*).
- global_cc: This function contains any global variable. Particularly, the user should define some required variables which we discuss in the following paragraph.
- post_processing_cc: This is an action to a post processing button. As we discussed, the postprocessing shows the missing edges in the nested dissection module.

A sample source code showing the previous functions for the column compression algorithm is as follows.

```
var reference_cc = function () {
    reference_text= "H. M. Buecker, M. A. Rostami, M. Luelfesmann : " +
        "An interactive educational module illustrating sparse matrix
            compression via graph coloring.";
    reference_url= "10.1109/ICL.2013.6644591";
};

var global_cc = function () {
    graph_format="cig";
    colors = range(0,22);
    chart_yaxis1_text = "Number of colors";
    chart_group5_text = 'Number of colors';
    start_matrix = "nestedDissection3.mtx";
    animation = true;
};
```

```

var column_compression = function() {
    var ns = neighbors(current);
    var col_ns = get_colors(ns);
    var new_col = min(diff(colors, col_ns));
    color_column(current, new_col);
    color_vertex(current, new_col);
    if (get_colored_vertices().length == currentg.vertices.length) {
     gather_round_data(min(diff(colors, get_colors(get_colored_vertices())))
        );
     round_completed();
    }
};

var post_processing_cc = function () {

};
```

# 5 Conclusion and future work

This dissertation is to develop the new ideas in combinatorial scientific computing mixing AD and preconditioning which we explained in Section 2.2. In Chapter 3, we introduce new coloring heuristics to increase the number of potentially and additionally required elements while the number of colors remains almost the same. Additionally, in the same chapter, we discussed a new heuristic for the coloring restricted to the diagonal elements. We implemented all these heuristics in our software *PreCol* which is explained in Section 3.4.

In Section 4, we develop an interactive educational module for teaching the concepts of combinatorial scientific computing. In this chapter, we discussed our previous publications [16, 17, 18, 19, 20] and also the new features which we have not published yet. There is still room for new ideas in this software. Beside developing new modules, we need to improve the usability and extensibility of the software. Also, we need an extensive evaluation of the software to find the new ways of further developments.

Although we proposed different heuristics, many dimensions of the problem can be improved. We considered only the ILU preconditioning with the natural ordering. A future work is to replace ILU with other preconditioning techniques. For example, the approximate inverse preconditioning (AINV) [66] might be a suitable candidate which produces no fill-in. Also, the connection of combinatorial techniques in support theory for preconditioning [67, 68] could be explored. Another future work is to consider blocks of submatrices for coloring instead of just a complete row or column? Steihaug and Hossain [69] discuss this idea for blocks of rows and the same column intersection graph as before and show it has advantages in the unidirectional coloring. A first improvement is to search for a similar approach in the bidirectional coloring and restricted coloring. Finally, a new area is to look at the same ideas for the hypergraph model for fine-grained coloring.

# Bibliography

[1] A. Griewank and A. Walther, *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*, 2nd ed. Philadelphia, PA: SIAM, 2008.

[2] L. B. Rall, *Automatic Differentiation: Techniques and Applications*, ser. LNCS. Berlin: Springer, 1981, vol. 120.

[3] A. Curtis, M. Powell, and J. Reid, “On the estimation of sparse Jacobian matrices,” *IMA Journal of Applied Mathematics (Institute of Mathematics and Its Applications)*, vol. 13, no. 1, pp. 117–119, 1974.

[4] T. F. Coleman and J. J. Moré, “Estimation of sparse Jacobian matrices and graph coloring problems,” *SIAM Journal on Numerical Analysis*, vol. 20, no. 1, pp. 187–209, 1983.

[5] J. J. E. Dennis and T. Steihaug, “On the successive projections approach to least-squares problems,” *SIAM Journal on Numerical Analysis*, vol. 23, no. 4, pp. 717–733, 1986. [Online]. Available: https://doi.org/10.1137/0723047

[6] T. F. Coleman and A. Verma, “Structure and efficient Jacobian calculation,” in *Computational Differentiation: Techniques, Applications, and Tools*, M. Berz, C. Bischof, G. Corliss, and A. Griewank, Eds. Philadelphia, PA: SIAM, 1996, pp. 149–159.

[7] A. S. Hossain and T. Steihaug, “Computing a sparse Jacobian matrix by rows and columns,” *Optimization Methods & Software*, vol. 10, pp. 33–48, 1998.

[8] S. Hossain and T. Steihaug, “Graph models and their efficient implementation for sparse Jacobian matrix determination,” *Discrete Applied Mathematics*, vol. 161, pp. 1747–1754, 2013.

[9] M. Hasan, S. Hossain, A. I. Khan, N. H. Mithila, and A. H. Suny, *DSJM: A Software Toolkit for Direct Determination of Sparse Jacobian Matrices*. Cham: Springer International Publishing, 2016, pp. 275–283. [Online]. Available: https://doi.org/10.1007/978-3-319-42432-3_34

[10] S. Hossain and T. Steihaug, “Optimal direct determination of sparse Jacobian matrices,” *Optimization Methods and Software*, vol. 28, no. 6, pp. 1218–1232, 2013.

[11] A. H. Gebremedhin, F. Manne, and A. Pothen, “What color is your Jacobian? Graph coloring for computing derivatives,” *SIAM Review*, vol. 47, pp. 629–705, 2005.

[12] A. Calotoiu, “Bipartite Graph Coloring for Compressed Sparse Jacobian Computation,” Master’s thesis, University Politehnica of Bucharest, Bucharest, Romania, 2009, prepared in Department of Computer Science, RWTH Aachen University.

[13] M. Lülfesmann, “Graphfärbung zur partiellen Berechnung von Jacobi-Matrizen,” Master’s thesis, Department of Computer Science, RWTH Aachen University, Aachen, Germany, 2006.

[14] M. Lülfesmann, “Full and partial Jacobian computation via graph coloring: Algorithms and applications,” Dissertation, Department of Computer Science, RWTH Aachen University, 2012. [Online]. Available: http://darwin.bth.rwth-aachen.de/opus3/volltexte/2012/4112/

[15] H. M. Bücker, M. Lülfesmann, and M. A. Rostami, “Enabling implicit time integration for compressible flows by partial coloring: A case study of a semi-matrix-free preconditioning technique,” in *2016 Proceedings of the Seventh SIAM Workshop on Combinatorial Scientific Computing, Albuquerque, New Mexico, USA, October 10–12*, A. H. Gebremedhin, E. G. Boman, and B. Ucar, Eds. Philadelphia, PA, USA: SIAM, 2016, pp. 23–32.

[16] H. M. Bücker, M. A. Rostami, and M. Lülfesmann, “An interactive educational module illustrating sparse matrix compression via graph coloring,” in *2013 International Conference on Interactive Collaborative Learning (ICL), Proceedings of the 16th International Conference on Interactive Collaborative Learning, Kazan, Russia, September 25–27, 2013.* Piscataway, NJ: IEEE, 2013, pp. 330–335.

[17] M. A. Rostami and H. M. Bücker, “Interactive educational modules illustrating sparse matrix computations and their corresponding graph problems,” in *Informatiktage 2014, Fachwissenschaftlicher Informatik-Kongress, 27. und 28. März 2014, Hasso Plattner Institut der Universität Potsdam*, ser. GI-Edition: Lecture Notes in Informatics (LNI) – Seminars, G. für Informatik, Ed. Bonn: Köllen Druck+Verlag GmbH, 2014, vol. S–13, pp. 253–256.

[18] H. M. Bücker and M. A. Rostami, “Interactively exploring the connection between nested dissection orderings for parallel Cholesky factorization and vertex separators,” in *IEEE 28th International Parallel and Distributed Processing Symposium, IPDPS 2014 Workshops, Phoenix, Arizona, USA, May 19–23, 2014.* Los Alamitos, CA, USA: IEEE Computer Society, 2014, pp. 1122–1129.

[19] H. M. Bücker and M. A. Rostami, “Interactively exploring the connection between bidirectional compression and star bicoloring,” in *International Conference on Computational Science, ICCS 2015 — Computational Science at the Gates of Nature, Reykjavík, Iceland, June 1–3, 2015*, ser. Procedia Computer Science, S. Koziel, L. Leifsson, M. Lees, V. V. Krzhizhanovskaya, J. Dongarra, and P. M. A. Sloot, Eds., vol. 51. Elsevier, 2015, pp. 1917–1926.

[20] M. A. Rostami and H. M. Bücker, “An educational module illustrating how sparse matrix-vector multiplication on parallel processors connects to graph partitioning,” in *Euro-Par 2015: Parallel Processing Workshops, Euro-Par 2015 International Workshops, Vienna, Austria, August 24–28, 2015, Revised Selected Papers*, ser. Lecture Notes in Computer Science, S. Hunold, A. Costan, D. Giménez, A. Iosup, L. Ricci, M. E. Gómez Requena, V. Scarano, A. L. Varbanescu, S. L. Scott, S. Lankes, J. Weidendorfer, and M. Alexander, Eds., vol. 9523. Cham, Switzerland: Springer, 2015, pp. 135–146.

[21] T. F. Coleman and A. Verma, “The efficient computation of sparse Jacobian matrices using automatic differentiation,” *SIAM Journal on Scientific Computing*, vol. 19, no. 4, pp. 1210–1233, 1998.

[22] D. Juedes and J. Jones, “Coloring Jacobians revisited: a new algorithm for star and acyclic bicoloring,” *Optimization Methods & Software*, vol. 27, no. 2, pp. 295–309, 2012.

[23] Y. Saad, *Iterative Methods for Sparse Linear Systems*, 2nd ed. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2003.

[24] M. Benzi, “Preconditioning techniques for large linear systems: A survey,” *Journal of Computational Physics*, vol. 182, no. 2, pp. 418–477, 2002.

[25] K. J. Cullum and M. Tůma, “Matrix-free preconditioning using partial matrix estimation,” *BIT Numerical Mathematics*, vol. 46, no. 4, pp. 711–729, 2006. [Online]. Available: http://dx.doi.org/10.1007/s10543-006-0094-8

[26] D. Hysom and A. Pothen, “Level-based incomplete LU factorization: Graph model and algorithms,” Lawrence Livermore National Labs, East Lansing, Michigan, Tech. Rep. Tech Report UCRL-JC-150789, November 2002.

[27] R. M. Karp, “Reproducibility among combinatorial problems,” in *Complexity of Computer Computations*, R. E. Miller and J. W. Thatcher, Eds. New York: Plenum Press, 1972, pp. 85–103.

[28] W. Hasenplaugh, T. Kaler, T. B. Schardl, and C. E. Leiserson, “Ordering heuristics for parallel graph coloring,” in *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA ’14. New York, NY, USA: ACM, 2014, pp. 166–177. [Online]. Available: http://doi.acm.org/10.1145/2612669.2612697

[29] D. W. Matula and L. L. Beck, “Smallest-last ordering and clustering and graph coloring algorithms,” *J. ACM*, vol. 30, no. 3, pp. 417–427, Jul. 1983. [Online]. Available: http://doi.acm.org/10.1145/2402.322385

[30] D. Brélaz, “New methods to color the vertices of a graph,” *Commun. ACM*, vol. 22, no. 4, pp. 251–256, Apr. 1979. [Online]. Available: http://doi.acm.org/10.1145/359094.359101

[31] D. J. A. Welsh and M. B. Powell, "An upper bound for the chromatic number of a graph and its application to timetabling problems," *The Computer Journal*, vol. 10, no. 1, pp. 85–86, 1967.

[32] T. F. Coleman and J. J. Moré, "Estimation of sparse Jacobian matrices and graph coloring problems," *Numerical Analysis*, vol. 20, pp. 187–209, 1983.

[33] T. A. Davis and Y. Hu, "The university of Florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 1:1–1:25, Dec. 2011.

[34] Ü. V. Çatalyürek, J. Feo, A. H. Gebremedhin, M. Halappanavar, and A. Pothen, "Graph coloring algorithms for muti-core and massively multithreaded architectures," *CoRR*, vol. abs/1205.3809, 2012. [Online]. Available: http://arxiv.org/abs/1205.3809

[35] G. Rokos, G. Gorman, and P. H. Kelly, *A Fast and Scalable Graph Coloring Algorithm for Multi-core and Many-core Architectures.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 414–425. [Online]. Available: http://dx.doi.org/10.1007/978-3-662-48096-0_32

[36] H. Büsing, J. Willkomm, C. H. Bischof, and C. Clauser, "Using exact Jacobians in an implicit Newton method for solving multiphase flow in porous media," *International Journal of Computational Science and Engineering*, vol. 9, no. 5/6, pp. 499–508, 2014. [Online]. Available: http://dx.doi.org/10.1504/IJCSE.2014.064535

[37] C. H. Bischof, H. M. Bücker, B. Lang, A. Rasch, and A. Vehreschild, "Combining source transformation and operator overloading techniques to compute derivatives for MATLAB programs," in *Proceedings of the Second IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2002).* Los Alamitos, CA, USA: IEEE Computer Society, 2002, pp. 65–72.

[38] J. Willkomm, C. H. Bischof, and H. M. Bücker, "A new user interface for ADiMat: Toward accurate and efficient derivatives of Matlab programs with ease of use," *International Journal of Computational Science and Engineering*, vol. 9, no. 5/6, pp. 408–415, 2014.

[39] A. Bondy and U. S. R. Murty, *Graph Theory*, ser. Graduate Texts in Mathematics. Springer, 2008.

[40] M. Wilson, *Imperfect C++: Practical Solutions for Real-Life Programming.* Addison-Wesley, 2004.

[41] R. F. Boisvert, R. Pozo, and K. Remington, "The matrix market exchange formats: Initial design," National Institute of Standards and Technology, Gaithersburg, MD, USA, Tech. Rep. Technical Report NITSTIR 5935, December 1996.

[42] *The Boost Graph Library: User Guide and Reference Manual.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[43] B. Sutherland, *Chapter 2: Modern C++.* Berkeley, CA: Apress, 2015, pp. 17–58. [Online]. Available: http://dx.doi.org/10.1007/978-1-4842-0157-2_2

[44] J. Singler and B. Konsik, "The GNU libstdc++ parallel mode: Software engineering considerations," in *Proceedings of the 1st International Workshop on Multicore Software Engineering*, ser. IWMSE '08. New York, NY, USA: ACM, 2008, pp. 15–22. [Online]. Available: http://doi.acm.org/10.1145/1370082.1370089

[45] M. A. Rostami, H. M. Bücker, and A. Azadi, "Illustrating a graph coloring algorithm based on the principle of inclusion and exclusion using GraphTea," in *Open Learning and Teaching in Educational Communities, Proceedings of 9th European Conference on Technology Enhanced Learning, EC-TEL 2014, Graz, Austria, September 16–19, 2014*, ser. Lecture Notes in Computer Science, C. Rensing, S. de Freitas, T. Ley, and P. J. Muñoz Merino, Eds., vol. 8719. Cham, Switzerland: Springer, 2014, pp. 514–517.

[46] M. A. Rostami, A. Azadi, and M. Seydi, "Graphtea: Interactive graph self-teaching tool," in *Communications, Circuits and Educational Technologies, Proceedings of the 2014 International Conference on Education and Educational Technologies II (EET'14), Prague, Czech Republic, April 2–4, 2014*, P. Dondon, B. K. Bose, D. S. Naidu, I. Rudas, and S. Kartalopoulos, Eds. EUROPMENT, 2014, pp. 48–51.

[47] A. Schliep and W. Hochstättler, "Developing Gato and CATBox with Python: Teaching graph algorithms through visualization and experimentation," *Multimedia Tools for Communicating Mathematics*, pp. 291–310, 2002.

[48] Y. Carbonneaux, J.-M. Laborde, and R. M. Madani, "CABRI-Graph: A tool for research and teaching in graph theory," in *Graph Drawing: Proceedings of the Symposium on Graph Drawing, GD '95, Passau, Germany, September 20–22, 1995*, ser. LNCS, F. J. Brandenburg, Ed., vol. 1027. Berlin: Springer, 1996, pp. 123–126.

[49] D. Auber *et al.*, "The Tulip 3 framework: A scalable software library for information visualization applications based on relational data," Research Centre Bordeaux–Sud-Ouest, INRIA, Research Report RR–7860, Jan. 2012. [Online]. Available: http://hal.archives-ouvertes.fr/hal-00659880

[50] A. Lambert and D. Auber, "Graph analysis and visualization with Tulip-Python," in *EuroSciPy 2012 — 5th European meeting on Python in Science*, Bruxelles, Belgique, 2012. [Online]. Available: http://hal.archives-ouvertes.fr/hal-00744969

[51] M. T. Heath, *Scientific Computing: An Introductory Survey*, 2nd ed. McGraw-Hill, 2002.

[52] C. B. Moler, *Numerical Computing with MATLAB.* Philadelphia, PA, USA: SIAM, 2004.

[53] S. Deterding, M. Sicart, L. Nacke, K. O'Hara, and D. Dixon, "Gamification: Using game-design elements in non-gaming contexts," in *CHI '11 Extended Abstracts on Human Factors in Computing Systems*, ser. CHI EA '11. New York, NY, USA: ACM, 2011, pp. 2425–2428. [Online]. Available: http://doi.acm.org/10.1145/1979742.1979575

[54] S. Deterding, D. Dixon, R. Khaled, and L. Nacke, "From game design elements to gamefulness: Defining "gamification"," in *Proceedings of the 15th International Academic MindTrek Conference: Envisioning Future Media Environments*, ser. MindTrek '11. New York, NY, USA: ACM, 2011, pp. 9–15.

[55] S. Leutenegger and J. Edgington, "A games first approach to teaching introductory programming," in *Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education*, ser. SIGCSE '07. New York, NY, USA: ACM, 2007, pp. 115–118. [Online]. Available: http://doi.acm.org/10.1145/1227310.1227352

[56] A. I. Wang, "Extensive evaluation of using a game project in a software architecture course," *Trans. Comput. Educ.*, vol. 11, no. 1, pp. 5:1–5:28, Feb. 2011. [Online]. Available: http://doi.acm.org/1921607.1921612

[57] L. Hakulinen, "Using serious games in computer science education," in *Proceedings of the 11th Koli Calling International Conference on Computing Education Research*, ser. Koli Calling '11. New York, NY, USA: ACM, 2011, pp. 83–88. [Online]. Available: http://doi.acm.org/10.1145/2094131.2094147

[58] A. Schäfer, J. Holz, T. Leonhardt, U. Schroeder, P. Brauner, and M. Ziefle, "From boring to scoring – a collaborative serious game for learning and practicing mathematical logic for computer science education," *Computer Science Education*, vol. 23, no. 2, pp. 87–111, 2013.

[59] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP–Completeness.* San Francisco: Freeman, 1979.

[60] M. Lülfesmann, S. R. Leßenich, and H. M. Bücker, "Interactively exploring elimination orderings in symbolic sparse Cholesky factorization," in *International Conference on Computational Science, ICCS 2010*, ser. Procedia Computer Science, vol. 1(1). Elsevier, 2010, pp. 867–874.

[61] A. A. Hagberg, D. A. Schult, and P. J. Swart, "Exploring network structure, dynamics, and function using NetworkX," in *Proceedings of the 7th Python in Science Conference*, G. Varoquaux, T. Vaught, and J. Millman, Eds., Pasadena, CA USA, 2008, pp. 11–15.

[62] J. D. Hunter, "Matplotlib: A 2D graphics environment," *Computing in Science Engineering*, vol. 9, no. 3, pp. 90–95, 2007.

[63] E. Jones *et al.*, "SciPy: Open source scientific tools for Python," 2014, http://www.scipy.org.

[64] Apache Software Foundation, "Mod_python module," 2013, http://www.modpython.org.

[65] A. Osmani, *Learning JavaScript Design Patterns*, ser. JavaScript and jQuery developer's guide. O'Reilly Media, Incorporated, 2012. [Online]. Available: https://books.google.de/books?id=JYPEgK-1bZoC

[66] M. Benzi and M. Tuma, "A sparse approximate inverse preconditioner for nonsymmetric linear systems," *SIAM Journal on Scientific Computing*, vol. 19, no. 3, pp. 968–994, 1998.

[67] K. Gremban, "Combinatorial preconditioners for sparse, symmetric, diagonally dominant linear systems," Ph.D. dissertation, Carnegie Mellon University, Pittsburgh, October 1996, cMU CS Tech Report CMU-CS-96-123.

[68] E. G. Boman and B. Hendrickson, "Support theory for preconditioning," *SIAM Journal on Matrix Analysis and Applications*, vol. 25, no. 3, pp. 694–717, 2003. [Online]. Available: https://doi.org/10.1137/S0895479801390637

[69] T. Steihaug and A. K. M. S. Hossain, "Graph coloring and the estimation of sparse Jacobian matrices with segmented columns," Department of Informatics, University of Bergen, Technical Report 72, 1997.

# Appendix

## A.1 Comparing the computations of Algorithm 3.1 and Algorithm 3.2

Here, we illustrate the figures for the comparison of Algorithm 3.1 and Algorithm 3.2. Each figure contains three computations: potentially required elements in the top figure, additionally required elements in the middle figure, and the number of colors in the bottom figure. Figure A.1, Figure A.2, and Figure A.3 are for the computation for the natural ordering, the LFO ordering, and the SLO ordering, respectively. Also, Figure A.4 shows this comparison for the matrix *crystm01* with the natural ordering.

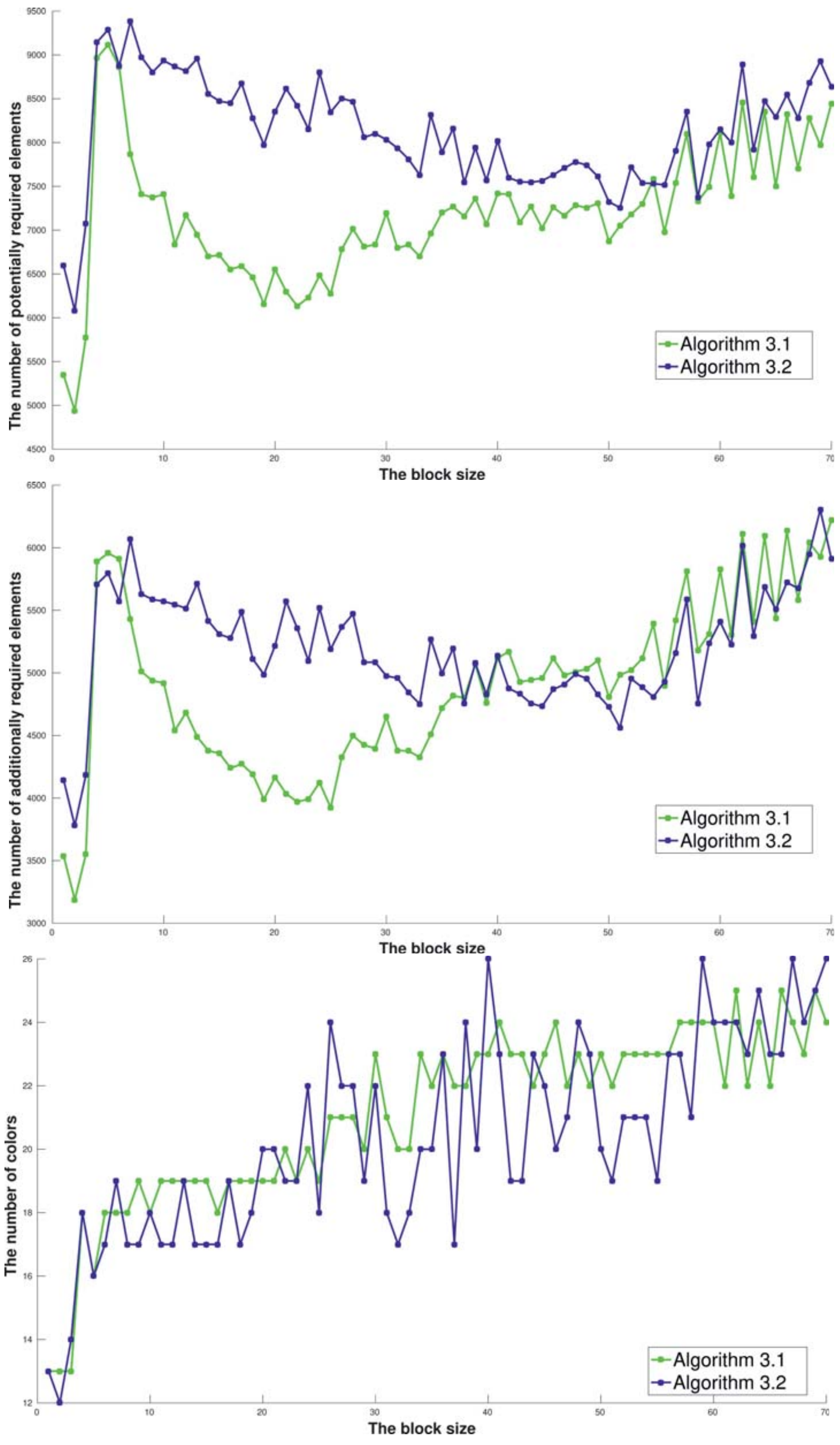


Figure A.1: The computation carried out the matrix *ex33* and for the natural ordering.

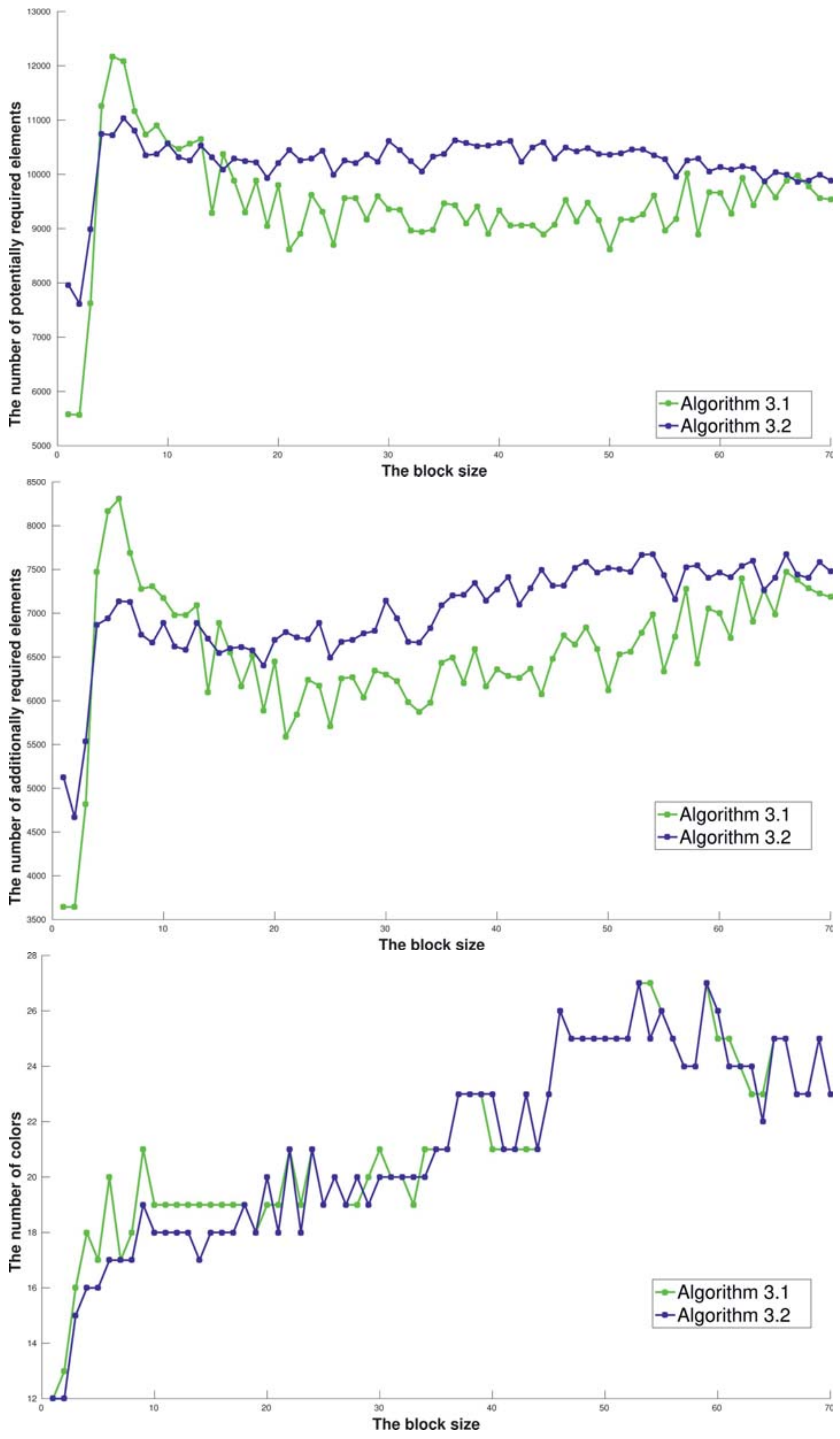


Figure A.2: The computation carried out the matrix *ex33* and for the LFO ordering.

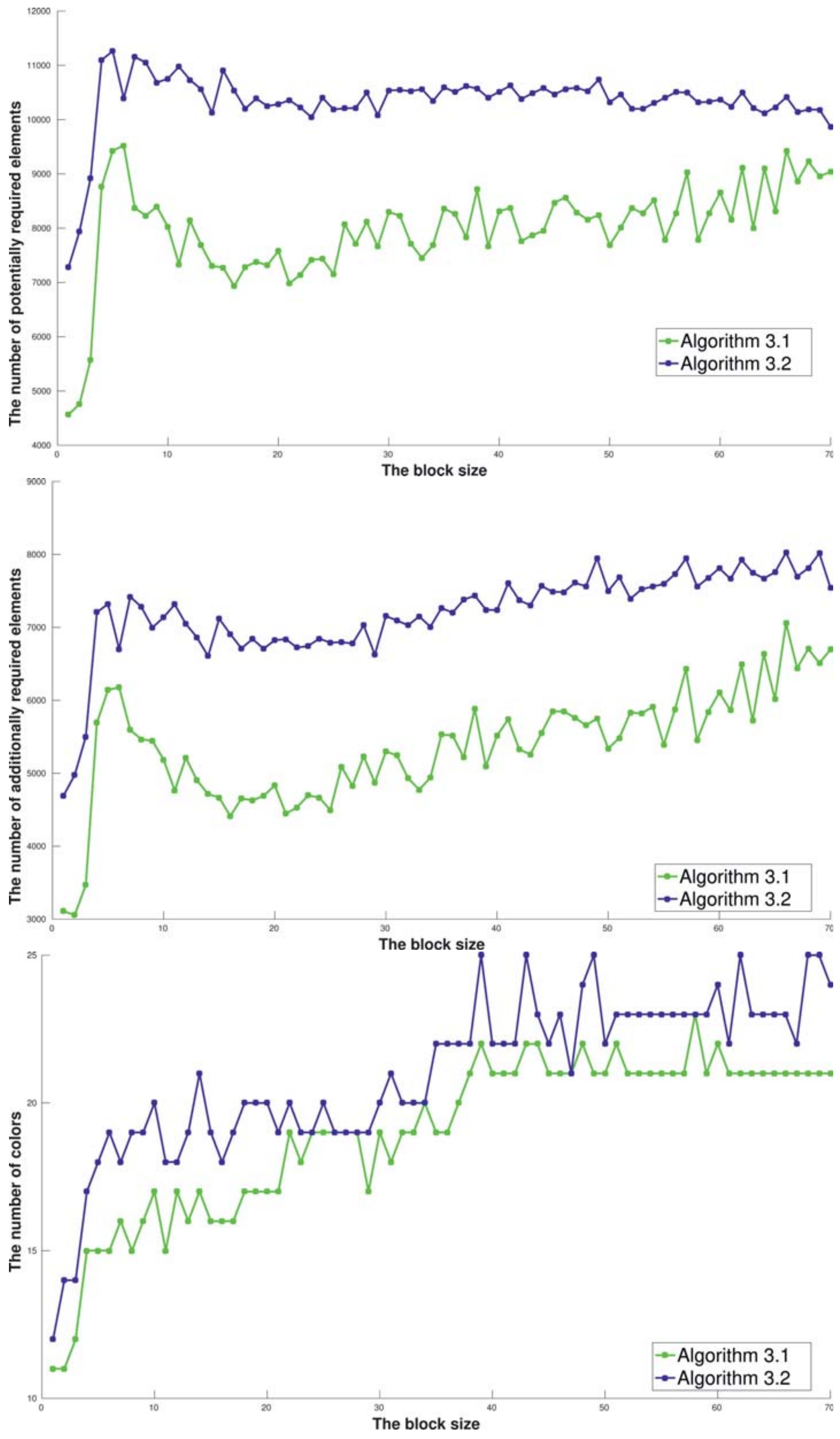


Figure A.3: The computation carried out the matrix *ex33* and for the SLO ordering.

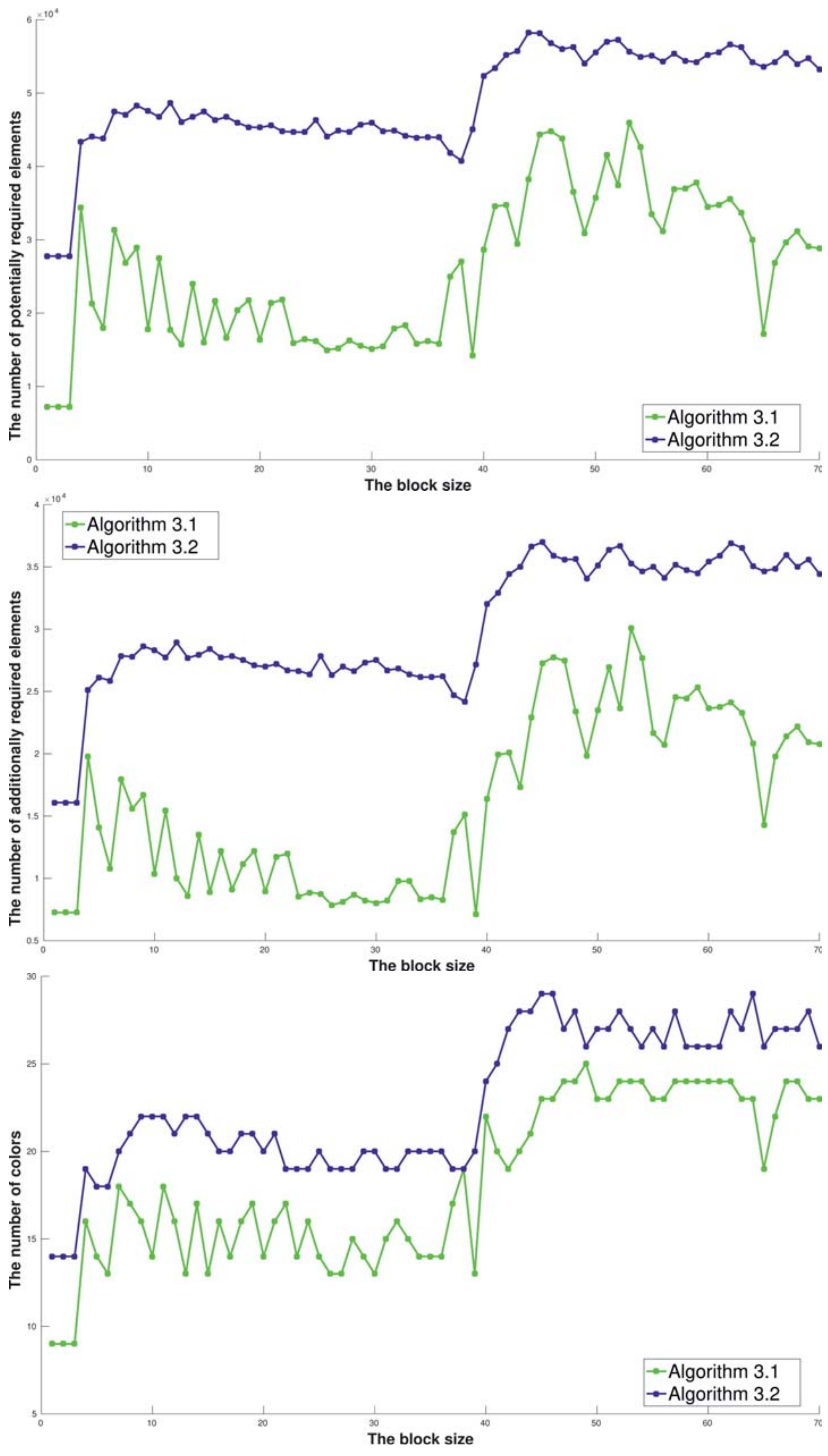


Figure A.4: The computation carried out the matrix *crystm01* and for the natural ordering.

## A.2 Comparing the computations of Algorithm 3.2 and Algorithm 3.4

Table A.1 shows the comparison of the number of colors and the number of potentially and additionally required elements produced by Algorithm 3.2 and Algorithm 3.4.

| Matrix (NAT) | $\lvert R_a \rvert$ | | $\lvert R_p \rvert$ | | $\lvert \Phi \rvert$ | |
|---|---|---|---|---|---|---|
| | Alg.3.2 | Alg.3.4 | Alg.3.2 | Alg.3.4 | Alg.3.2 | Alg.3.4 |
| *steam1.mtx* | 630 | 64 | 786 | 64 | 10 | 7 |
| *steam2.mtx* | 1400 | 240 | 1880 | 240 | 17 | 9 |
| *nos3.mtx* | 4296 | 1106 | 6756 | 1638 | 19 | 13 |
| *ex7.mtx* | 25054 | 29174 | 34954 | 38554 | 55 | 56 |
| *ex33.mtx* | 5572 | 4920 | 8934 | 7408 | 18 | 18 |
| *crystm01.mtx* | 28318 | 10388 | 47556 | 17822 | 22 | 14 |
| *coater1.mtx* | 7448 | 7684 | 11558 | 11722 | 27 | 28 |
| *pesa.mtx* | 33094 | 31010 | 41154 | 36972 | 13 | 12 |

| Matrix (LFO) | $\lvert R_a \rvert$ | | $\lvert R_p \rvert$ | | $\lvert \Phi \rvert$ | |
|---|---|---|---|---|---|---|
| | Alg.3.2 | Alg.3.4 | Alg.3.2 | Alg.3.4 | Alg.3.2 | Alg.3.4 |
| *steam1.mtx* | 666 | 64 | 1048 | 64 | 12 | 7 |
| *steam2.mtx* | 1248 | 240 | 2624 | 240 | 17 | 9 |
| *nos3.mtx* | 4442 | 1246 | 6882 | 1880 | 21 | 16 |
| *ex7.mtx* | 24060 | 28904 | 33426 | 37080 | 53 | 59 |
| *ex33.mtx* | 6888 | 7170 | 10564 | 10574 | 18 | 19 |
| *crystm01.mtx* | 21194 | 12256 | 36634 | 20326 | 17 | 17 |
| *coater1.mtx* | 7536 | 7410 | 11512 | 11312 | 24 | 24 |
| *pesa.mtx* | 31884 | 31790 | 41676 | 42490 | 11 | 11 |

| Matrix (SLO) | $\lvert R_a \rvert$ | | $\lvert R_p \rvert$ | | $\lvert \Phi \rvert$ | |
|---|---|---|---|---|---|---|
| | Alg.3.2 | Alg.3.4 | Alg.3.2 | Alg.3.4 | Alg.3.2 | Alg.3.4 |
| *steam1.mtx* | 754 | 64 | 1294 | 64 | 14 | 7 |
| *steam2.mtx* | 1912 | 240 | 3192 | 240 | 17 | 9 |
| *nos3.mtx* | 4382 | 1132 | 6772 | 1682 | 21 | 13 |
| *ex7.mtx* | 24164 | 27044 | 34448 | 36486 | 55 | 51 |
| *ex33.mtx* | 7138 | 5186 | 10754 | 8024 | 20 | 17 |
| *crystm01.mtx* | 26782 | 14252 | 45166 | 24478 | 20 | 16 |
| *coater1.mtx* | 7878 | 7004 | 11702 | 10476 | 24 | 21 |
| *pesa.mtx* | 34044 | 29034 | 44624 | 39606 | 13 | 10 |

Table A.1: The comparison between the number of potentially and additionally required elements as well as the number of colors computed with Algorithm 3.4 and 3.2. The block size is fixed to 10. The orderings for coloring are (Top) the natural ordering, (Middle) LFO, and (Bottom) SLO.

## A.3 Comparing the computations of Algorithm 3.5 with different block sizes

Figure A.5 and Figure A.6 show the comparison of the number of colors and the number of additionally required elements produced by Algorithm 3.5 with different values of $\alpha$ and the varying block sizes.

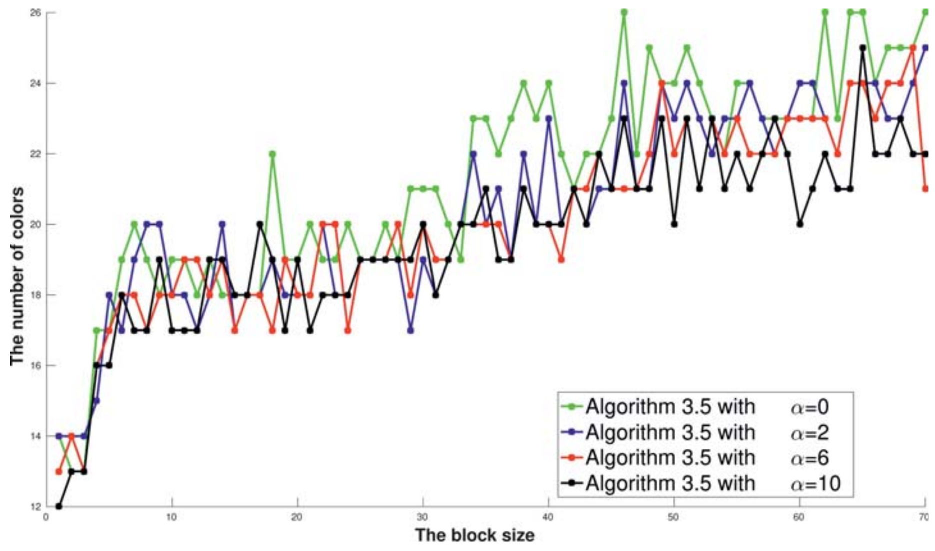


Figure A.5: The comparison of the number of colors in Algorithm 3.5 with $\alpha \in \{0, 2, 6, 10\}$ and the LFO ordering.

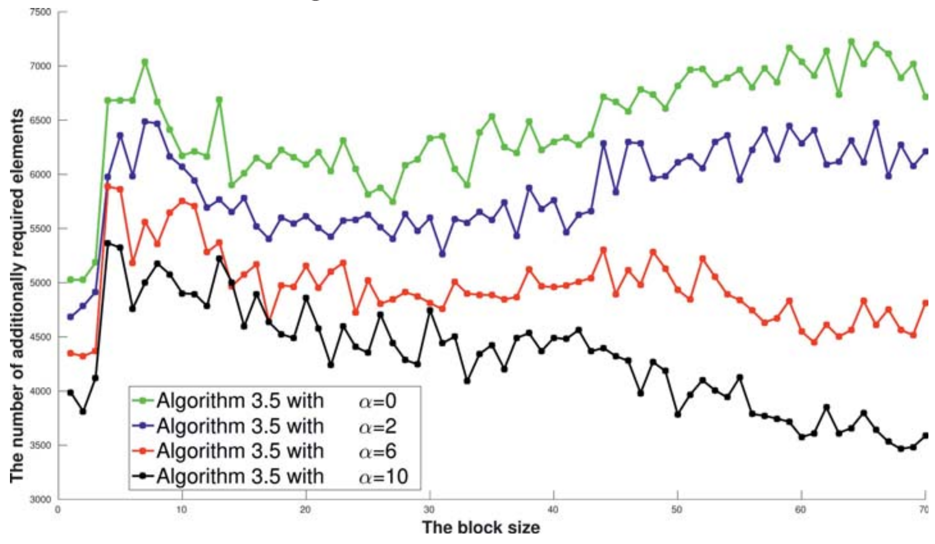


Figure A.6: The comparison of the number of additionally required elements in Algorithm 3.5 with $\alpha \in \{0, 2, 6, 10\}$ and the LFO ordering.