seit 1558

COMBINATORIAL SCIENTIFIC COMPUTING

Mohammad Ali Rostami
Supervisor: Prof. Martin Bücker

Faculty of Mathematics and Computer Science
Friedrich Schiller University Jena

September 8, 2016

Abstract

Solving problems originating from the real-world applications are often based on the solution to some linear equations containing a sparse Jacobian matrix. These sparse Jacobian matrices are most of the time huge. Hence, there is much research to exploit the sparsity structure and to decrease the amount of needed storage. In contrast to full Jacobian computation in which all elements are meant to be determined, partial Jacobian computation is looking at a subset of these elements. This property brings a lot of conventions leading to a faster and more efficient computation. Determining these nonzero elements in full or partial Jacobian computations by automatic differentiation techniques can be modeled as graph coloring in graph language.

Since a matrix-vector product (instead of the whole Jacobian) is needed in iterative solvers, automatic differentiation fits the best in the iterative solvers. On the other hand, the preconditioning techniques are used to improve the convergence of iterative solvers and need access to the nonzero elements. So, a sparsification applies on the Jacobian before computing the preconditioner. The elements produced from the sparsification are also considered as the required elements in the partial coloring. There is a procedure which selects a subset of the remaining nonrequired elements and adds to the sparsified matrix such that no fill-ins or increase of the number of colors happens. The current thesis first looks at different ways to optimize the processes of selecting elements. At the second part of this thesis, we introduce an interactive educational module to teach these tricky concepts in the classroom.

Zusammenfassung

Acknowledgements

This project would not have been possible without the support of many people. I must express my first gratitude towards *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. Besides, I would like to thank *Michael Lülfesmann* 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

Chapter 1

Introduction

Partial differential equations (PDE) are the most common models of the real-world problems. Iterative solvers are a solution to these linear systems which need a multiplication of Jacobian and a vector in each iteration. Many strategies were studied to compute these products efficiently. Automatic Differentiation [1, 2] computes not only the differentiation without numerical errors but also the matrix-vector multiplication automatically. Based on this ability of AD techniques, different methods are analyzed to reduce the memory that is needed to store the Jacobian matrix as well as speeding up the computation. These methods are generally as a part of a bigger area called combinatorial scientific computing.

On the other hand, another method to speed up the convergence of the linear solver is preconditioning. The preconditioning techniques need most of the time access to the elements of Jacobian which is in contrast to the requirements of automatic differentation. Applying a sparsification operator first on the Jacobian matrix and do the ILU preconditioning on the sparsified matrix would be a solution to the problem of determiniation of all elmenets. This processes becomes complete by trying to add more elements to the matrix without producing more fill-ins or colors. We introduce new heuristics which consider this elements in the time of coloring in favour of increasing them.

This dissertation is structured as follows. First, we go through basic definitions in Chapter 2 which are needed in other chapters. We discuss the known graph model for full Jacobian computation and partial Jacobian computation. A sparsification is introduced which helps in the formulation of mix of automatic differentation and preconditioning. Chapter 3 discusses

new heuristics for distance-2 coloring and star bicoloring. We also look at a simple parallelization technique for the coloring algorithm. also We look at the ordering for the ILU preconditioning. In this chapter, we restructure a previous software to be more readable and extensible and to contain new heuristic algorithm. Chapter 4 discuss an interactive educational module which is designed to teach the computational science algorithms in the classroom. Finally, the conclusion and future work come in Chapter 5.

Chapter 2

State of the Art

In this chapter, we would briefly discuss the already exisiting concepts which we need to understand the other parts of the paper. In each section of this chapter, we provide some references to read these concepts further in details. These concepts are introduced in three parts of full Jacobian computation in Section 2.1, partial Jacobian computation Section 2.2, and the preconditioning in Section 2.3.

2.1 Full Jacobian Computation

2.1.1 Problem

If there is a program which computes the function $f(x) : \mathbb{R}^n \to \mathbb{R}^m$ in 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 so-called forward mode of automatic differentiation creates a program automatically which computes the product of Jacobian matrix with some given so-called seed matrix V, i.e. JV. There is an inverse mode of automatic differentiation which computes the product WJ. These techniques of automatic differentiation do these computations without assembling the Jacobian J. Suppose the matrix V has c number of columns and the matrix W has r number of rows, the computational costs to compute these products would be 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 a suitable linear combination of the columns of J or the reverse mode with a suitable linear combination of the rows of J; see the survey [3].

The key idea behind this *unidirectional compression* is now illustrated for the forward mode. First, we present a definition as follows,

Definition 1 (Structurally Othogonality) *Let $J = [c_1, c_2, \ldots, c_n]$ denote the Jacobian matrix whose ith column is represented by the vector $c_i \in \mathbb{R}^m$. 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 combinations of structurally orthogonal columns of the seed matrix to reeduce the number of columns of the seed matrix. More precisely, a set S of structurally orthogonal columns can be represented by a single column of the product JV because the sum of these columns contains all the nonzero entries of all the columns in S. A set of structurally orthogonal rows is also represented by a single row in the product WJ. The computational cost then scales with the number of groups of structurally orthogonal columns or rows in the forward or reverse mode, respectively. Figure 2.1 (Left) and Figure 2.1 (Middle) shows two examples of the matrices which can be compressed efficiently by columns and rows, respectively. Next, consider the matrix C that has neither structurally orthogonal columns nor structurally orthogonal rows as in Figure 2.1 (Right). Therefore, there is no unidirectional compression of the matrix C, neither by c columns nor by r rows, that reduce c or r below the number of columns or rows. However, a linear combination of both, columns and rows, can be used to reduce the computational cost to a value below the size of matrix. Here, the columns and rows of a corresponding group are not necessarily structurally orthogonal. This technique in which the forward and reverse mode are used in a combined way is called *bidirectional compression*.

For general sparsity patterns, it is not always easy to figure out how to linearly combine columns and rows such that the computational cost is

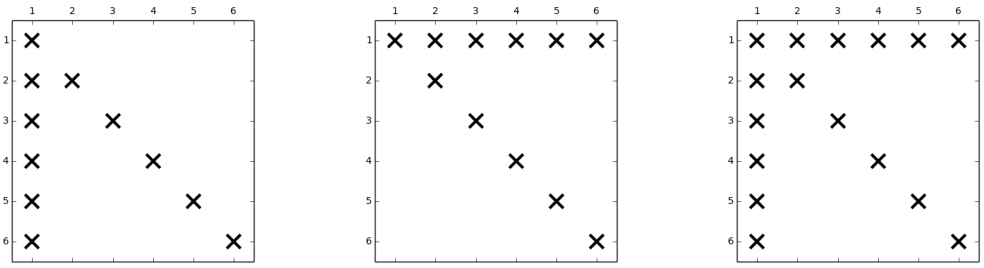


Figure 2.1: (Left) An example of a matrix which can be compressed efficiently by columns. (Middle) An example of a matrix which can be compressed efficiently by rows. (Right) An example of a matrix which cannot be compressed efficiently either by columns nor by rows.

minimized. Hence, we introduce the following combinatorial optimization problems that addresses this question of unidirectional or bidirectional compression. In practice, the solution of this problem will substantially reduce the computational cost for computing all nonzero elements of a large and sparse Jacobian.

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

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 $r \times m$ whose number of columns of V and 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.*

2.1.2 Combinatorial Model

We reformulate the scientific computing problems which we have discussed in Section 2.1.1. The new formulation is an equivanet problem defined on a carefully chosen graph model. The survey [3] 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 J consists of a set of vertices $V = \{v_1, v_2, \ldots, v_n\}$ whose vertex v_i represents the ith 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 have a nonzero element in at least a same row position.*

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

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

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 Φ of the column intersection graph G with a minimal number of colors.*

As it can be seen, there is an intimate connection between the sparse Jacobian computations described in Problem 1 and the graph coloring issues described by Problem 3. If a vertex represents a row and an edge represents structural non-orthogonality of two rows, a row-intersection graph can be adapted too.

Although, this model is convincing for the unidirectional compression, the bidirectional compression can not be computed in this model. A bidirectional compression needs the information of both rows and columns which is not encoded. A bipartite graph model is defined for this purpose as in [5, 6, 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 1 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 classic definition of neighbourhood.

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 mapping

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

that assigns to each vertex a color represented by an integer. Recall from the previous section that, in general, there can be columns or rows that are not chosen in the grouping at all. Therefore, the coloring Φ also involves a "neutral" color representing this "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 columns and rows. Since the overall cost is the sum of the costs associated to the forward mode and to 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. Recall from our example with the products CV and WC 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 problem MINIMUM BIDIRECTIONAL COMPRESSION asks for computing *all* nonzero elements, at least one vertex of every edge has to be colored with a non-neutral color.

3. Recall from the example that structural orthogonality is no longer required for grouping the rows and columns. However, there is still the following restriction. 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 matrix positions (i,k), (i,ℓ), 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,ℓ) is already determined by the reverse mode for row i the column vertex c_ℓ is assigned the neutral color. However, if (i,ℓ) is computed by the forward mode, a third (non-neutral) color is needed for c_ℓ. 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 to 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 coloring 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 [8].

2.2 Partial Jacobian Computation

2.2.1 Problem

Gebremedhin et al [3] introduced the concept of the partial Jacobian computation in which the assumption is that only a subset of Jacobian should be determined which is called the required elements. Lülfesmann [9] has studied more considerably in this area. Suppose the required elements are shown by R, the definition of *structurally orthogonality* is adapted for the partial coloring as follows,

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

The corresponding problem can be formulated as follows for either unidirectional or bidirectional compression,

Problem 5 (Minimum Restricted Unidirectional Compression) *Let J be a sparse $m \times n$ Jacobian matrix with known sparsity pattern and R be a proper subset of J. Find a seed matrix V of dimension $n \times c$ whose number of columns of V sums up to a minimal value, c, such that all nonzero elements of R also appear in the matrix-matrix product JV.*

Problem 6 (Minimum Restricted Bidirectional Compression) *Let J be a sparse $m \times n$ Jacobian matrix with known sparsity pattern and R be a proper subset of J. Find a pair of binary seed matrices V of dimension $n \times c$ and W of dimension $r \times m$ whose number of columns of V and 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.*

An equivalent graph-theoretical formulation of this problem is discussed in the next section.

2.2.2 Combinatorial Model

Based on [3, 9], the definitions of the full Jacobian colorings are adapted 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 $E_R \subset E$, then a mapping $\Phi : V_c \to \{0, 1, \ldots, p\}$ is a distance-2 coloring of G restricted to E_R if no column vertex gets a zero color and for every path (c_k, r_i, c_j) with $c_k \in V_c$, $\Phi(c_k) \neq \Phi(c_j)$.*

Definition 8 (Restricted star bicoloring) *Given a bipartite graph $G = (V_c \cup V_r, E)$ and a subset of required edges $E_R \subset E$, then a mapping $\Phi : V_c \cup V_r \to \{0, 1, \ldots, p\}$ 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 and edge in E_R recieves 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* $\Phi(r_i) = 0$ *and* $\Phi(c_j) = 0$, *then for every path* (c_k, r_i, c_j, r_l), $\Phi(c_k) \neq \Phi(c_j)$ *or* $\Phi(r_i) \neq \Phi(r_l)$

The optimization problems would be to find the minimum restricted distance-2 coloring and the minimum restricted star bicoloring.

2.2.3 Sparsification (Block Diagonal)

The third computational ingredient is sparsification of the coefficient matrix J. The sparsification of J is represented by the symbol $\rho(J)$. The idea is to select a subset of all nonzero elements of J, described by $\rho(J)$, and construct a preconditioner based on these selected nonzero elements [10]. These nonzero elements selected by $\rho(J)$ are called *required* nonzero elements. Throughout this paper we consider the special case of a sparsification in the form of $k \times k$ blocks on the main diagonal. That is, the pattern of the required nonzero elements is given by the pattern of all nonzero entries within these $k \times k$ blocks on the diagonal. If the block size k does not divide the matrix order n, we adapt the size of the last diagonal block to some smaller value such that the sum of all block sizes equals n. We formulate the new problem that represents the assembly of the semi-matrix-free approach with a minimal relative cost as follows.

Problem 7 (Block Seed) *Let* J *be a sparse* $n \times n$ *Jacobian matrix with known sparsity pattern and let* $\rho(J)$ *denote its sparsification using* $k \times k$ *blocks on the diagonal of* J. *Find a binary* $n \times p$ *seed matrix* S *with a minimal number of columns,* p, *such that all nonzero entries of* $\rho(J)$ *also appear in the compressed matrix* $J \cdot S$.

We now focus on reformulating this combinatorial problem from scientific computing in terms of an equivalent problem defined on a suitable graph model. Recall that the set of nonzero elements is divided into required and nonrequired elements. Two columns can be linearly combined without loosing information on the required elements as long as one of the following conditions is satisfied:

- There is no row position in which both columns have a nonzero element, whether required or nonrequired.
- There is one or more row positions in which both columns have a nonzero element and both these nonzeros in the same row are nonrequired elements.

Thus, the case where two columns can be assigned to the same column group is encoded by the following definition.

Definition 9 (Structurally ρ-Orthogonal) *A column $J(:, i)$ is structurally ρ-orthogonal to column $J(:, j)$ if and only if there is no row position ℓ in which $J(\ell, i)$ and $J(\ell, j)$ are nonzero elements and at least one of them belongs to the set of required element $\rho(J)$.*

We now construct a modified column intersection graph in which the set of vertices is the same as in Def. 2, but whose set of edges is defined differently.

Definition 10 (ρ-Column Intersection Graph) *The ρ-column intersection graph $G_\rho = (V, E_\rho)$ associated with a pair of $n \times n$ Jacobians J and $\rho(J)$ consists of a set of vertices $V = \{v_1, v_2, \ldots, v_n\}$ whose vertex v_i represents the ith column $J(:, i)$. Furthermore, there is an edge (v_i, v_j) in the set of edges E_ρ if and only if the columns $J(:, i)$ and $J(:, j)$ represented by v_i and v_j are not structurally ρ-orthogonal.*

So, the edge set E_ρ is constructed in such a way that columns represented by two vertices v_i and v_j need to be assigned to different groups if and only if $(v_i, v_j) \in E_\rho$. This constructions shows that Problem 7 is equivalent to the following coloring problem.

Problem 8 (Minimum Block Coloring) *Find a coloring Φ of the ρ-column intersection graph G_ρ with a minimal number of colors.*

A column of J without any required nonzero element is represented by a vertex to which no edge is incident in G_ρ. These isolated vertices does not need to be colored. So, in general, colors are assigned to a subset of the vertices in G_ρ.

Finally, an alternative formulation can be derived by using a bipartite graph model in which there is a vertex set for the rows and another vertex

set for the columns of the Jacobian. Using the more general bipartite graph model, Problem 8 could be reformulated as a minimum distance-2 coloring of a bipartite graph when restricted to the set of required edges. This is called partial coloring in [3].

2.3 Preconditioning

2.3.1 Problem

The computation of Jacobian that we discussed before are one part of solving PDEs. In general, we are considering the solution to the following system of linear equations,

$$J\mathbf{x} = \mathbf{b},$$

in which J is a large sparse non-symmetric non-singular Jacobian matrix. An approximative iterative solver is an effective solution since they have a faster convergence and need less memory. As we discussed, these solvers are matrix-free which makes AD as a suitable method of differentiation in this case.

Another way to speedup these iterators are the preconditioning techniques [11, 12]. Rather than solving the previous unpreconditioned system. we can consider solving the preconditioned system

$$M^{-1}J\mathbf{x} = M^{-1}\mathbf{b}, \tag{2.1}$$

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

$$M \approx J.$$

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. In [9], a novel approach is explained. We discuss this approach briefly here.

Common approaches to construct 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 in terms of computing time.

We discussed the sparsification operator $\rho(J)$ which selects a subset of the nonzeros of the Jacobian matrix J. It does not mean that the whole Jacobian is available but only the nonzero pattern of the matrix should be known known. Now, the preconditioner is built based on these selected nonzero elements [10]. These nonzero elements selected by $\rho(J)$ are called initially *required* nonzero elements and represented by R_i. An example would be the special case of a sparsification in the form of $k \times k$ blocks on the main diagonal.

We discussed the full and partial coloring algorithms regarding AD techniques. Now, the problem 8 can be solved for Jacobian restricted to the set of initially required nonzero elements R_i. The result of coloring would group columns and rows together. In this processes, the required elements would always be computed. However, the nonrequired elements would be divided into two sets of elements: the elements which are computed and the ones which would be eliminated. We can think of these computed nonzero elements as the byproduct of the coloring. Since the number of colors does not change, an idea would be to add this extra byproducts also to R_i. However, it could generate new fill-ins in the preconditioning step. So, we call these new set of byproduct elements as the potentially required elements R_p and computed as follows,

$$R_p \subset pat(A) - R_i \quad \text{so that} \quad |\Phi(R_i)| = |\Phi(R_i \cup R_p)|.$$

As we have discussed, an ILU preconditioner is applied to R_i which can produce a set of fill-in. We consider thte Different methods can be employed

to compute the preconditioner. A purely matrix version of the ILU preconditioning can be found in [13]. Hysom and Pothen [14] introduced first a graph model for the incomplete LU factorization in which the matrix is actually the adjacency matrix of this graph. It should be considered that the graph would be directed if the matrix is not symmetric. A concept of *fill path* is defined to characterize the fill-ins in preconditioning,

Definition 11 (Fill path) *A fill path is a path* $(v_i, ..., v_k, ..., v_j)$ *with* $k < min(i, j)$*. It means the index of the all inner nodes in a given ordering of vertices should be smaller than indices of the vertices* v_i *and* v_j*. Equivalently, a matrix element* (i, j) *is a fill-in 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 [14]. This parameter is used to filter the generated fill-ins. Actually this parameter is the length of the fill path. The level-based incomplete LU factorization the generated fill-ins are allowed to be considered only up to the level l.

Now, a maximum subset of potentially required elements is selected such that no new fill-ins are generated which is called additionally required elements R_a. These elements can be added to the initially required elements for further computation. The additionally required elements can be 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.

2.3.2 Combinatorial Model

The bipartite graph model presented in Definition 4 can be used to design new algorithms for finding the initially, potentially, and additionally required elements. Three subsets $E_i, E_p, E_a \subset E$ are considered for the initially, potentially, and additionally required elements, respectively. Given the sparsification operator $\rho$and the Jacobian matrix A, here is a list of steps in the computation of additionally required elements and the corresponding algorithm on the bipartite graph model (details can be found in [9],

- compute the initially required elements $R_i = \rho(A)$.
- setup the bipartite graph G from A and the subset $E_i \subset E$ fom R_i.

```
1 function pot_d2_color(G_b,E_i)
2   R_p = ∅
3   for (r_i,c_j) ∈ E_i with Φ(c_j) ≠ 0
4     for c_k ∈ N_1(r_i,c_j) with j ≠ k and (r_i,c_k) ∉ E_i
5       if Φ(c_j) = Φ(c_k)
6         continue
7     E_p = E_p \cup {(r_i,c_j)}
```

Algorithm 2.1: Find potentially required elements for distance-2 coloring

- compute the partial coloring the bipartite graph G restricted to E_i and save colors as the properties of vertices $\Phi_G(E_i)$
- find a subset of edges $E_p \subset E - E_i$ such that $\Phi_G(E_i) = \Phi_G(E_i \cup E_p)$
- find a subset of potentially required edges $E_a \subset E_p$ such that $SILUR(R_i) \cup R_a = SILU(R_i \cup R_a)$

an improved version We consider this new version to compute the additionally required elements throughout this thesis.

```
1  function pot_star_bicoloring(G_b, E_i)
2    E_p = ∅
3    for (r_i, c_j) ∈ E − E_i with Φ(r_i) ∉ 0 or Φ(c_j) ∉ 0
4      if (Φ(r_i) = 0)
5        for c_k ∈ N_1(r_i, G) with j ≠ k and (r_i, c_k) ∉ E_i
6          if (Φ(c_j) == Φ(c_k))
7            continue with the next edge (r_i, c_j) ∈ E − E_i
8
9      if (Φ(c_j) = 0)
10       for r_l ∈ N_1(c_j, G) with j ≠ l and (r_l, c_j) ∉ E_i
11         if (Φ(r_i) == Φ(r_l))
12           continue with the next edge (r_i, c_j) ∈ E − E_i
13
14     if (Φ(r_i) ≠ 0 and Φ(c_j) ≠ 0)
15       for c_k ∈ N_1(r_i, G) with i ∉ k
16         for r_l ∈ N_1(c_j, G) with j ∉ l
17           if (Φ(c_j) == Φ(c_k) and Φ(r_i) == Φ(r_l))
18             continue with the next edge (r_i, c_j) ∈ E − E_i
19
20     E_p = E_p ∪ (r_i, c_j)
```

Algorithm 2.2: Find potentially required elements for star bicoloring

```
1  function add(G_b, E_i, E_p)
2    E_a = ∅
3    for (r_i, c_j) ∈ E_p
4      if i > j
5        if ∃c_l ∈ N_1(r_j, G[E_i ∪ (E_f ∪ E_a)]) with l > j
6          continue with next edge (r_i, c_j) ∈ E_p
7      else if i > j
8        if ∃r_k ∈ N_1(c_i, G[E_i ∪ (E_f ∪ E_a)]) with k > i
9          continue with next edge (r_i, c_j) ∈ E_p
10     E_a = E_a ∪ (r_i, c_j)
```

Algorithm 2.3: Find additionally required elements

```
1  function add_improved(G_b, E_i, E_p)
2    E_a = ∅
3    do
4      for (r_i, c_j) ∈ E_p
5        if i > j
6          if ∃c_l ∈ N_1(r_j, G[E_i ∪ (E_f ∪ E_a)]) with l > j
7            continue with next edge (r_i, c_j) ∈ E_p
8        else if i > j
9          if ∃r_k ∈ N_1(c_i, G[E_i ∪ (E_f ∪ E_a)]) with k > i
10           continue with next edge (r_i, c_j) ∈ E_p
11       E_a = E_a ∪ (r_i, c_j)
12       E_p = E_p − (r_i, c_j)
13   while |E_a| is increased in the last iteration
```

Algorithm 2.4: Find additionally required elements

Chapter 3

Preconditioning and Coloring

The different steps of compting the additionally required elements are discussed in Section 2.3.2. Here, we look at effects of ordering on the number of fill-ins in Section 3.1 and how it can be fit into the automatic differentiation. In Section 3.2, we discuss different strategies to increase the number of additionally required elements as well as first ideas toward parallelizations.

Lülfesmann and Bücker [9] started a package for computation of the uni- and bidirectional (partial-)coloring as well as the partial computation and its relative preconditioning which we discussed in 2.3. We improved the software in various ways that are explained in Section 3.3. Finally, a MATLAB interface and a JAVA interface for this software package is introduced in Section 3.3.2.

Throught this thesis, we did experiments on some matrices from the Flordia sparse matrix collection [15]. Table 3.1 shows these matrices togther with their information.

Name	Size	Nonzeros
nos3	960×960	15844
gyro_m	17361×17361	340431
ex33	1733×1733	23922

Table 3.1: The experiments are done these matrices from the Florida sparse matrix collection throught this thesis.

3.1 Experiments on Effects of Orderings

In the idea of mixing the preconditioning and AD which we have discussed in Section 2.3, the ILU preconditioning is computed using the natural ordering of the given matrix. However, it is a well known fact that the number of fill-ins is highly dependent on the ordering. So, we can improve the number of fill-ins by carefully choosing an ordering. On the other hand, we compute the Jacobian matrix by AD techniques in which the matrix computed in a specific ordering. We need a reformulation to fit these two computation from ILU preconditioning and the automatic differentation.

As we discussed in the iterative solvers, like BICGSTAB, we always need to have a matrix-vector product like Az. This suits the AD behaviour which gets a seed matrix, here the vector z, and computes the matrix-vector product. A reordering of the matrix for ILU preconditioning needs a consideration also in the seed matrix. Without reordering, a preconditioning would look like as,

$$Ax = b \rightarrow M^{-1}Ax = M^{-1}b.$$

We would add the reordering to this equation results in the following equations,

$$\begin{aligned} Ax = b \rightarrow M^{-1}Ax &= M^{-1}b \\ P^T M^{-1} P P^T A P P^T x &= P^T M^{-1} P P^T b \\ (P^T M^{-1} P)(P^T A P) P^T x &= (P^T M^{-1} P) P^T b \\ (P^T M P)^{-1}(P^T A P) P^T x &= (P^T M P)^{-1} P^T b \\ \tilde{M}^{-1}\tilde{A}\tilde{x} &= \tilde{M}^{-1}\tilde{b}. \end{aligned}$$

As this equation shows, we need a reordering in the matrix A if we have the reordering in the preconditioner. So, the matrix-vector product $\tilde{A}.\tilde{x} = P^T A P x$ should be computed instead of $A.x$.

Let the function *ad* to compute the automatic differentiation and the fuction *bicgstab* to compute a step of BICGSTAB iterative solver, we modify the seed matrix and the return result of this function to adapt the reordering as follows,

$$\begin{aligned} z &= P\tilde{x} \\ APz &= ad(f, z, ...) \\ \tilde{x} &= bicgstab(P^T A P z, ...) \end{aligned}$$

in which the seed matrix $P\tilde{x}$ is used instead of x. After computation of AD, the resulting matrix-vector product APz, should be multiplied by P^T from the left before continuing by the computations regarding BICGSTAB.

Here, we investigate the preconditioning based on the incomplete LU factorization (ILU) [16]. Between different models of ILU, we consider the level-based ILU factorization here. We use a graph model for ILU instead of the current matrix model to have a unified work on graphs in the implemented library. This graph model is based on the proposed model in [14]. If the given matrix is asymmetric, we put a vertex for each row. We also connect the vertex i to the vertex j with an directed edge if the corresponding element (i, j) in matrix in nonzero. If the matrix is symmetric, these edges are undirected. This means the graph is a simple graph and the given matrix is actually the adjacency matrix of the graph. Now, we look at the effects of preordering for the vertices of the given preconditioning graph to reduce fill-ins in ILU factorization. Later, we would study further how this fill-in reduction would increase the additionally required elements.

As same as coloring algorithms, finding an ILU factorization with the minimum fill-ins is also an NP-complete problem. There are a lot of literature considering this problem[17, 18, 19, 20]. Again, the ILU factorization is computed in a specific order which is essential in the minimum fill-ins. Here we bring an example to show how the order affects the ILU factorization and the number of fill-ins. For example, let's consider first the following matrix,

$$\begin{bmatrix} 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \end{bmatrix}.$$

We setup a graph for ILU preconditoning. The graph for preconditoning would be an undirected graph since the matrix is symmetric. For example, the graph in Figure 3.1 is computed from the previous matrix. Here, we illustrated the computation of Cholesky factorization step by step. Figure 3.1 computes the Cholesky factorization in which the order of vertices are the numbering written on the vertices as labels. The ordering in Figure 3.1 is actually a worst-case ordering that produces 6 fill-ins. These fill-ins are illustrated as dotted lines.

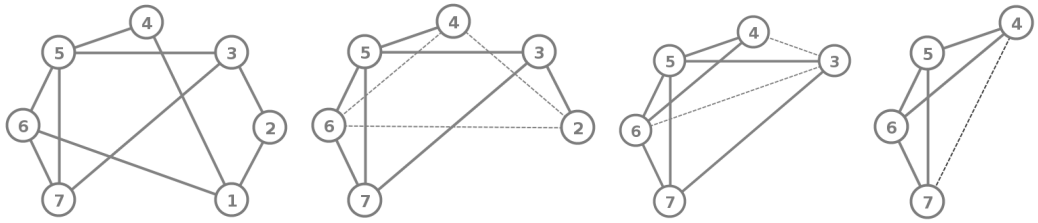


Figure 3.1: A worst case ordering generates 6 fill-ins. The ordering here is the numbering which visualized as labels.

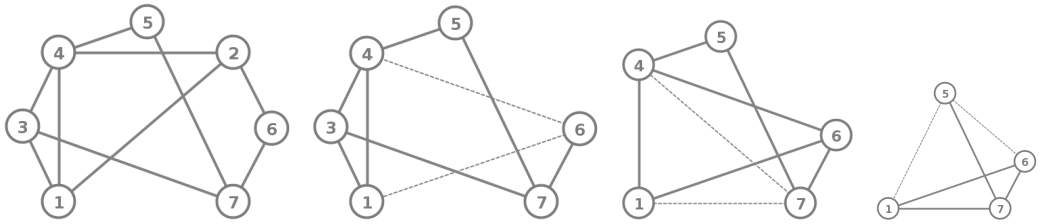


Figure 3.2: The order of elimination $2, 3, 4$ generates 5 fill-ins.

Figure 3.2 shows how the new ordering produces 5 fill-ins when the ordering is $2, 3, 4$. However, the best ordering produces only 3 fill-ins as Figure 3.3 shows when the ordering is $2, 5, 3$.

There are various ordering which were studied for coloring heuristics throughout years. As we discussed in the previous section, there are different ordering for coloring like *LFO*, *IDO*, and *SLO*. In addition to these orderings, we consider three other orderings for preconditioning:

- Natural Ordering (Nat): This ordering is based on the ordering of the matrix rows and columns given as input.

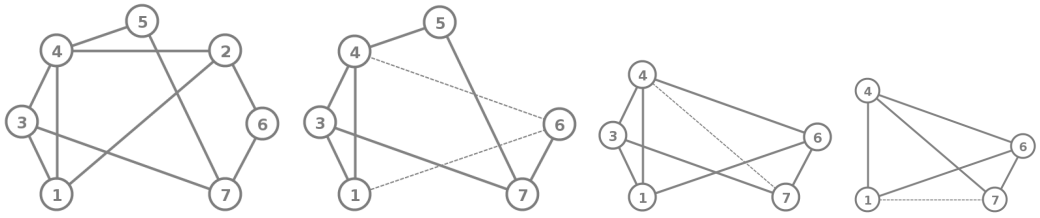


Figure 3.3: The order of elimination $2, 5, 3$ generates 3 fill-ins.

Orders	Colors	Fill-ins
Nat	15	70
Min	15	52
Metis	15	52

Orders	Colors	Fill-ins
Nat	34	8760
Min	34	8414
Metis	34	642

Table 3.2: The number of fill-ins compared based on the different orderings for the graph vertices. The number of colors remains the same since we change only the ordering for ILU factorization. The left table is the computation for the matrix *nos3.mtx* and the right one the computation is for the matrix*gyro_m*. Both are from the Florida sparse matrix collections.

- Minimum Degree Ordering (Min): The one generates a list of vertices which are sorted based on the degree from minimum to maximum.
- Metis Ordering[21, 22] (Metis): This ordering is generated by the well known tool *Metis* which is a fill-reducing ordering. It is computed by the algorithm of nested dissection.

The Table 3.2 shows how different ordering generates different fill-ins required elements. The number of fill-ins compared based on the different orderings for the graph vertices. The number of colors remains the same since we change only the ordering for ILU factorization. The left table is the computation for the matrix *nos3.mtx* and the right one is the computation for the matrix*gyro_m*. Both are from the Florida sparse matrix collections.

These results are computed on the matrix *nos3.mtx* and the matrix *gyro_m* from the Florida sparse matrix collections. The block size is chosen to be 50 for the matrix *gyro_m* and 15 for the matrix *nos3*. Also, the coloring algorithm would be a one-sided partial coloring. As it can be seen in this table, the fill-reducing Metis ordering generates the minimum fill-in. Another observation is that the Metis and Min ordering are generating almost the same number of fill-ins for the small matrices. It is only for the big matrices which the efficiency of Metis ordering can be seen.

This difference in number of fill-ins can affect the convergence speed relatively. Both Figure 3.4 and Figure 3.5 show the convergence history of the bicgstab solvers on the matrix *nos3*. The three line charts are the convergence history of the solver with no preconditioning, ILU preconditioning with block-diagonal sparsification, and ILU preconditioning with block-diagonal sparsification together with the found additionally required elements, respectively. Here, the block size is chosen to be 10 and the level of ILU factorization

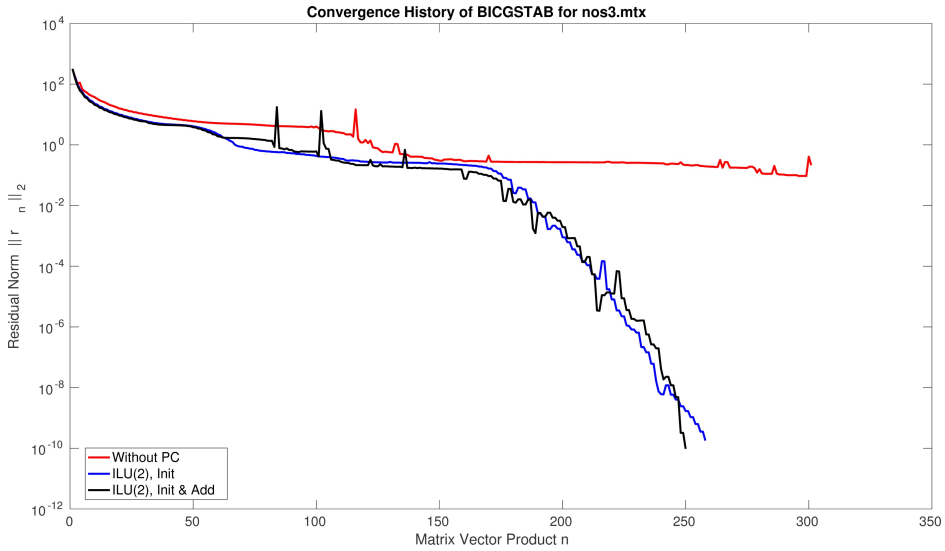


Figure 3.4: Natural ordering results in a worst convergence.

is 10. Clearly, Figure 3.5 has a better convergence rate in the chart in which the additionally required elements are added in comparison with the same chart in Figure 3.4.

So far, we decided for the level of ILU factorization arbitrarily. Here, we want to see the actual influence of the level parameter on the fill-ins which is visualized in Figure 3.6,

On the other hand, the coloring has its influence on the number of additionally required elements. Table 3.3

Here, we did not change any settings other than the ordering of coloring. The ordering for ILU preconditioning is fixed to the natural ordering. The block size is 50 and 10 and the ILU level is 5 and 2 for two matrices *gyro_m* and *nos3*, respectively. The matrix is *gyro_m*.

It can be seen that the number of additionally required elements increases when the number of colors decreases. This can be understood as the number of potentially required elements are chosen from the non-required elements and added to the required elements such that number of colors does not increase. Hence, if the number of colors are more, the freedom of choosing elements gets higher. As two tables 3.2 and 3.3 shows, both the ordering of coloring and the ordering of preconditioning affects the number of additionally required elements. Although smaller number of fill-in increases the

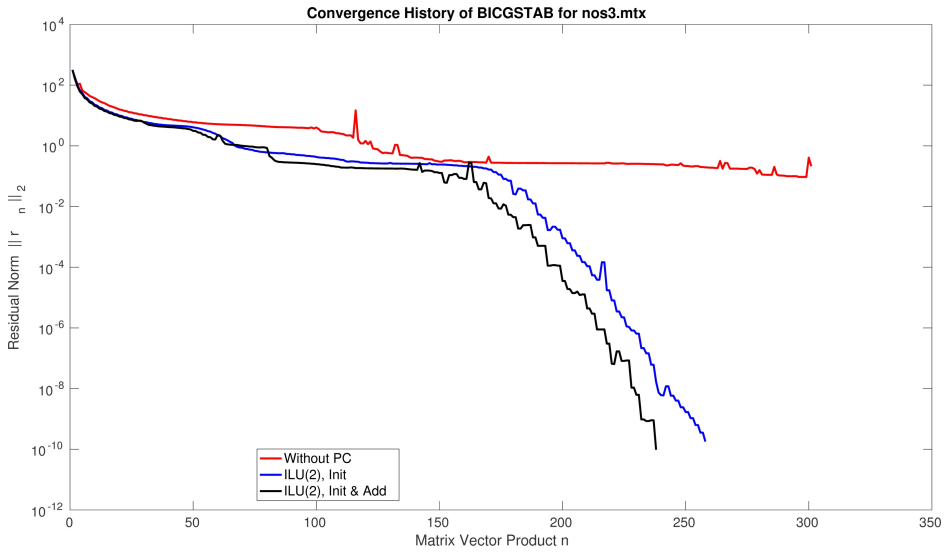


Figure 3.5: Metis ordering results in a better convergence.

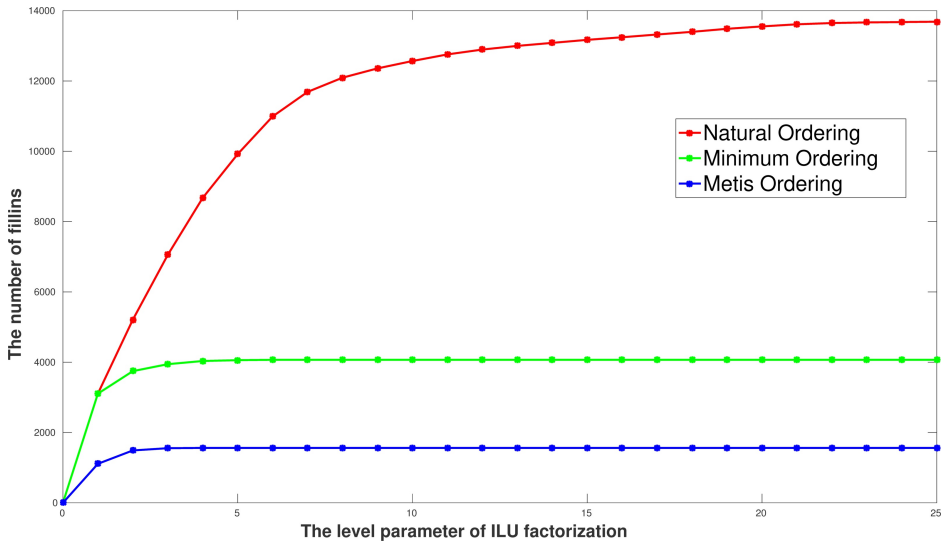


Figure 3.6: The influence of the level parameter for ILU on the number of fill-ins while three different orderings are employed. The block size is fixed to 100. The matrix is *ex33* and the level parameter changes from 0 to 25. The ordering for coloring is *LFO*.

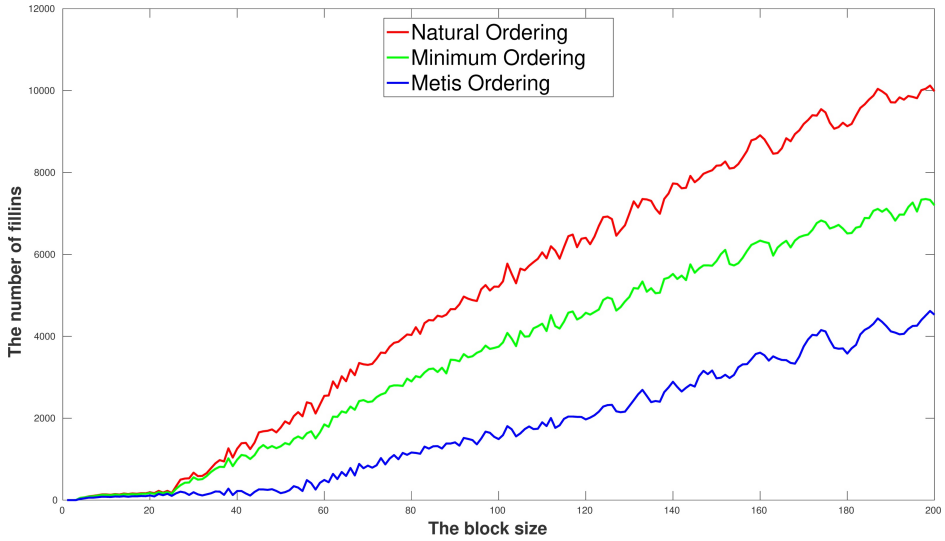


Figure 3.7: The influence of the block size for sparsification on the number of fill-ins while three different orderings are employed. The block size is fixed to 100. The matrix is *ex33* and the block size changes from 1 to 200. The ordering for coloring is *LFO*.

Ordering	Colors	Fill-ins	Additionally required elements
LFO	34	8760	66554
SLO	33	8760	57760
IDO	35	8760	66588
Ordering	Colors	Fill-ins	Additionally required elements
LFO	15	50	394
SLO	12	50	384
IDO	16	50	436

Table 3.3: The effect of coloring on the number of additionally required elements computed for two matrix *gyro_m* and *nos3*.

number of additionally required elements, the smaller number of colors decreases the number of additionally required elements. This is in contrast to the minimum number of colors which we are searching for, in the automatic differentiation.

3.2 New heuristics for coloring

In partial coloring, we compute the coloring focusing on required nonzero elements. However, there are other nonrequired nonzero elements which are also computed as a by-product of the computation of required elements. These are important elements for the determination of potentially required elements and additionally required elements. Look at the following example,

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

in which the symbol r stands for the required element, the symbol $*$ stands for the other nonzero elements, and the number 0 denotes the actual zero elements. If the first and second column are colored with the same color, you will get the nonzero at position $(3, 1)$ as a by-product. However, there are certain degrees of freedom. In this example, you could also assign the same color to columns 1 and 3 in which no nonzero element in the last row will be computed as a by-product. This leads to the question of maximizing the number of nonzero elements that are computed as a by-product.

Let's look at the same story differently. The columns 2 or 3 could have the same color with the first column to compute the required elements. (Of course, 2 and 3 are not allowed to have the same color because the required elements would sum up.) However, the number of nonrequired elements which survive would increase if we color the first and second column with the same color and the other one with the other color. Now, the question is: can we save more nonrequired elements in the process of coloring? Can we maximize a term like $\alpha C + \beta N$ where alpha and beta are weighting coefficients, C is the coloring number, and N is the number of survived nonrequired elements? Here, we introduce a new heuristic which increases the number of survived nonrequired elements such that the number of colors remains almost the same.

3.2.1 Distance-2 coloring

Since finding an exact coloring is a known NP-complete problem [23], different heuristics are employed to obtain an estimation of minimum coloring. The greedy coloring is a widely used coloring. The greedy coloring algorithm computes a reasonable coloring [24]. The greedy coloring algorithm for a simple graph is given in a pseudocode as follow,

```
1 function d2_color(G,V)
2   for v ∈ V with Φ(v) = 0
3     forbiddenColors[0] = v
4     if ∃n ∈ N(v) : (v, n) ∈ E_i
5       for n ∈ N(v)
6         if Φ(n) ≠ 0
7           forbiddenColors[Φ(n)] = v
8     Φ(v) = min{a > 0 : forbiddenColor[a] ≠ v
```

Algorithm 3.1: New coloring heuristc.

The computational complexity of the greedy coloring is low. It is also easy to implement. Among the various versions of this coloring, the version showed here has the computational complexity of $\mathcal{O}(n+m)$ in which n is the number of vertices and m is the number of edges [25]. We need to iterate over all vertices and check the incidence condition for all edges connected to each vertex. It can also be written as $\mathcal{O}(n + \Delta)$ since we need only to check neighbors of each vertex.

Algorithm 3.1 depends on the vertex ordering. Hence, there are many publications on how to choose a suitable ordering heuristics for a serial or parallel version of coloring [25, 26, 27]. Various orderings are inquired for coloring heuristics throughout years. Here is a list of orderings for coloring,

- Largest-First Ordering (LFO) [28] chooses a vertex with the minimum degree in each step.
- Incidence-degree Ordering (IDO) [29] chooses first the vertex with maximum degree in G, namely v. Then, it selects the matrix with the maximal degree in the subgraph induced by $V(G) - v$. It means the vertex with the maximum incidence degree is selected.
- Saturation-degree Ordering (SDO) [30] chooses first the vertex with the maximum degree in G, namely v. Then, it chooses the vertex with the maximum saturation degree in V. The saturation degree of the vertex v is the number of different colored vertices in the neighbors of v.

- Smallest last ordering (SLO) [25] makes a set of vertices $v_1, v_2, ..., v_n$ in a so-called smallest-last order whenever v_i has minimum degree in the maximal subgraph on the vertices $v_1, v_2, ..., v_i$ for all i.

The computation of this ordering can have a higher complexity than $\mathcal{O}(n+m)$ or to have the same complexity. For example, the *SLO* ordering has the same time complexity [25] although it has the different space complexity. These various orderings can be used to improve the results of coloring. In this coloring, the assumption is that each column is available one at a time. Hence, we can not compare the number of nonrequired elements in the processes of coloring. For this purpose, we introduce a new heuristic coloring algorithm.

Our approach in this new heuristic coloring algorithm is based on finding an independent set. A subset of the vertices $S \subset V$ is called an independent set if no edge between any of these vertices exists, i.e.,

$$\forall u, v \in S : (u, v) \notin E.$$

For a given vertex v, we show an independent set containing v by I_v. We also define the number of nonrequired elements which survives when two columns i and j are compressed as $N_{nreq}(i, j)$. Now, we compute $N_{nreq}(u, v)$ for each element $u \in I_v$ and color a vertex from the set I_v with the maximum value of N_{nreq} beside coloring the vertex v in each step and with the same color as v. The following pseudocode shows this approach,

The time complexity of this new heuristic is estimated as follow. We need to go through all vertices and find the maximum independent set containing each vertex. Then, we need to find all nonrequired elements which can be survived. It is to be of the order $\mathcal{O}(n^2)$ although the comptation takes shorter actually since we color two vertices in each step.

Figure 3.8 shows the number of additionally required elements computed by this new approach, compared with the greedy algorithm. Clearly, the new approach computes more additionally required elements als than the greedy approach. This is an interesting result since the number of colors remains almost the same as it can be seen in Figure 3.9. Both computations is carried out on the matrix *ex33* from the Florida sparse matrix collection with 1733 rows, 1733 columns, and 23922 nonzeros. The block size changes from 1 to 70. The number of colors in Figure 3.9 for the greedy coloring is the minimum value between different orderings *LFO*, *SLO*, and *IDO*.

So far, we select a vertex with the maximum nonrequired elements. However, it can happen that we have more than one vertex with the maximum

```
1  function d2_color_nreq(G,V)
2    for v ∈ V with Φ(v) = 0
3      forbiddenColors[0] = v
4      if ∃n ∈ N(v) : (v,n) ∈ E_i
5        for n ∈ N(v)
6          if Φ(n) ≠ 0
7            forbiddenColors[Φ(n)] = v
8
9      Φ(v) = min{a > 0 : forbiddenColor[a] ≠ v
10     I_v = {u ∈ V_c : u ∉ N_2(v)}
11     nreq_v = {n ∈ N(v) : (n,v) ∉ E_i}
12     for i ∈ I_v
13       nreq_i = {n ∈ N(i) : (n,i) ∉ E_i}
14       req_i = {n ∈ N(i) : (n,i) ∈ E_i}
15       mapVtoNreq = mapVtoNreq ∪ (i, count(nreq_i − nreq_v))
16     maxs = {a ∈ keys(map) : map[a] = max(values(mapVtoNreq))}
17     Φ(maxs[0]) = Φ(v)
```

Algorithm 3.2: New coloring heuristc.

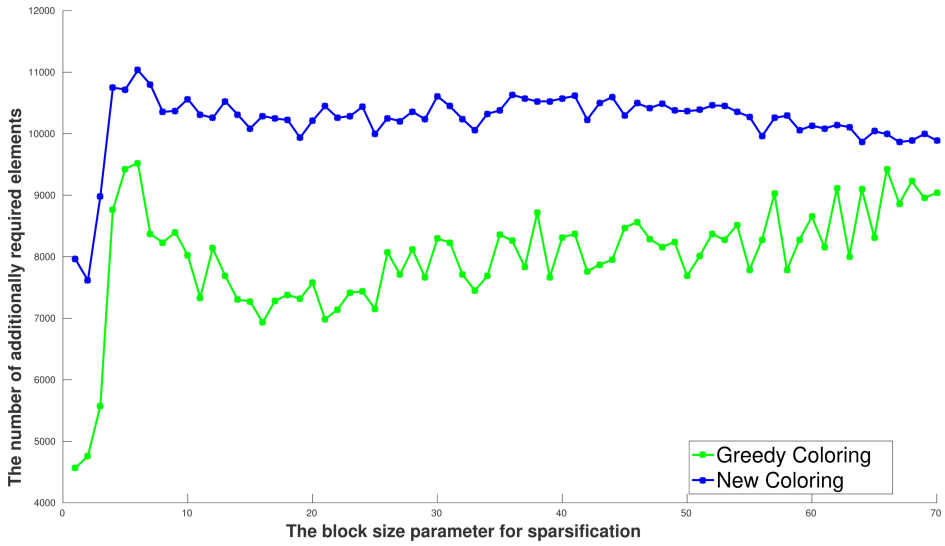


Figure 3.8: The number of colors in the new heuristic coloring compared with the greedy algorithm. The computation is carried out on the matrix *ex33*.

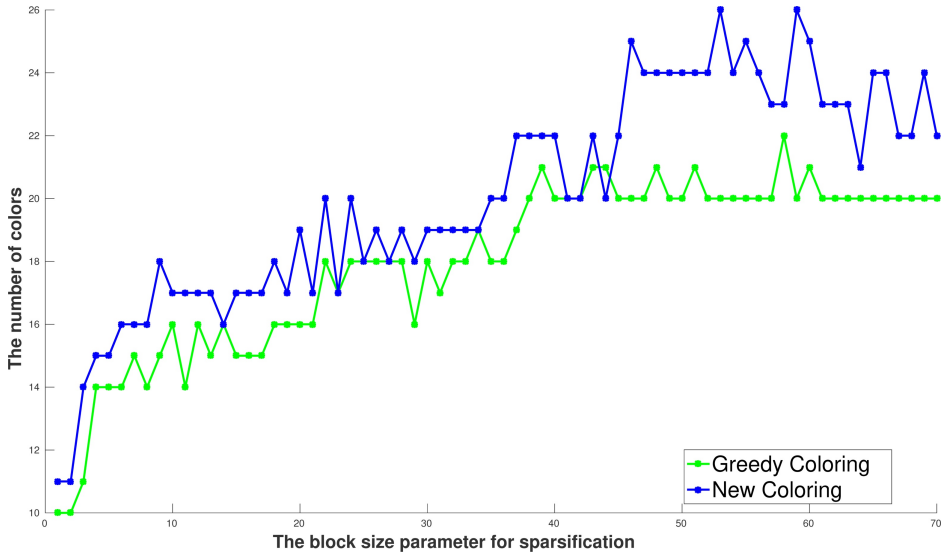


Figure 3.9: The number of additionally required elements in the new heuristic coloring compared with the greedy algorithm. The computation is carried out on the matrix *ex33*.

nonrequired elements in each step. Suppose we are looking at vertex v and two vertices a and b have maximum values of $N_nreq(v,a) = N_nreq(v,b)$. Adding both elements v_i and v_j would not always gives us more additionally required elements because the value of $N_nreq(a,b)$ can be very small. Therefore, an improvement is to select one vertex from this set of maximum values such that the number of additionally required elements increase. As we have seen, a potential required element can be an additional required element if the addition of this element to the initially required elements does not introduce any new fill-in. Having less number of initially required elements in a column would decrease the possibility of producing more fill-ins since it would decrease the whole number of fill paths.

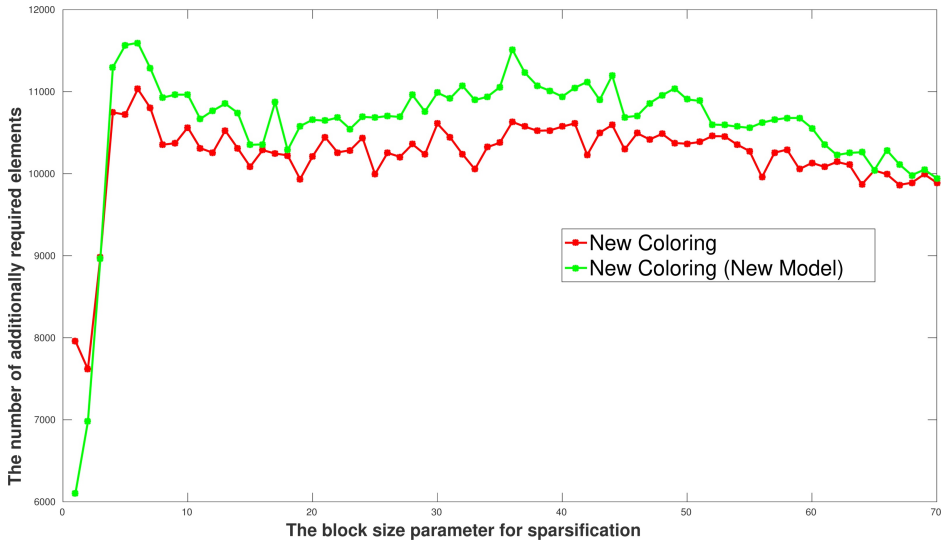
So, the previous algorithm is modified as follow,

The time complexity is again $\mathcal{O}(n^2)$ since the sort of map happens in the lower order of $\mathcal{O}(nlog(n))$. Figure 3.2.1 shows how the number of additionally required elements are increased by the previous strategy.

Our algorithm is to color the columns such that the number of colors remain almost the same while the number of additionally required elements are increased. What if one can have some control over colors too. A good

```
1  function d2_color_nreq(G,V)
2    for v ∈ V with Φ(v) = 0
3      forbiddenColors[0] = v
4      if ∃n ∈ N(v) : (v,n) ∈ E_i
5        for n ∈ N(v)
6          if Φ(n) ≠ 0
7            forbiddenColors[Φ(n)] = v
8
9      Φ(v) = min{a > 0 : forbiddenColor[a] ≠ v}
10     I_v = {u ∈ V_c : u ∉ N_2(v)}
11     nreq_v = {n ∈ N(v) : (n,v) ∉ E_i}
12     for i ∈ I_v
13       nreq_i = {n ∈ N(i) : (n,i) ∉ E_i}
14       req_i = {n ∈ N(i) : (n,i) ∈ E_i}
15       mapVtoNreq = mapVtoNreq ∪ (i, count(nreq_i − nreq_v))
16       mapVtoreq = map ∪ (i, count(req_i))
17     maxs = {a ∈ keys(map) : map[a] = max(values(mapVtoNreq))}
18     minReq = {a ∈ maxs : mapVtoreq[a] = min(values(mapVtoreq))}
19     Φ(minReq) = Φ(v)
```

Algorithm 3.3: New coloring heuristc increasing the number of additionally required elements.



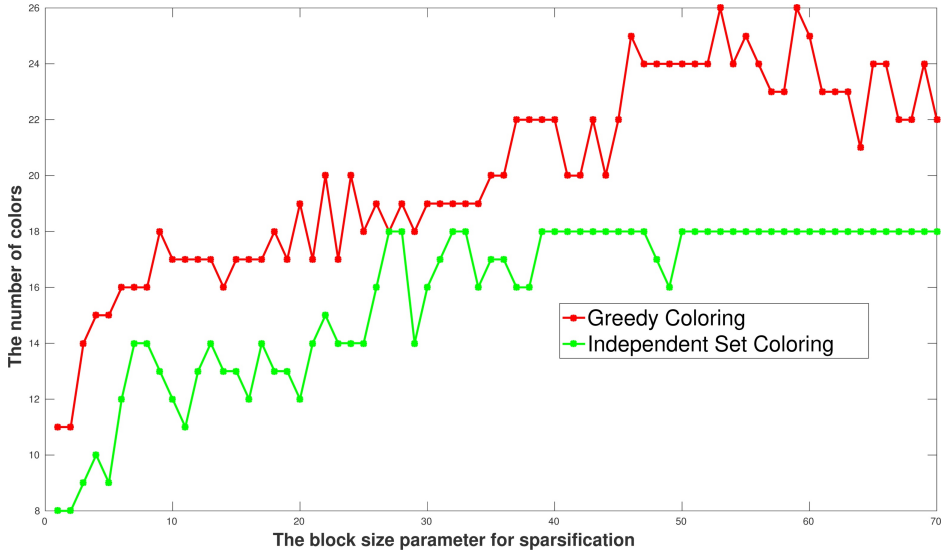


Figure 3.10: The comparison of an algorithm which color the independent sets first and the greedy algorithm with respect to the number of colors.

heuristic for coloring is to color independent sets in each step. For example, Figure 3.10 shows a comparison between a greedy algorithm and an algorithm which colors the independent set first. As you can see the independent set algorithm produces definitely less number of colors. However, it does not perform good in the case of additionally required elements as in Figure 3.11.

It is a logical result since we saw that selecting two columns with the maximum number of nonrequired elements does not give us always more number of additionally required elements. So, we propose to choose only column with the maximum number of nonrequired elements and choose α other columns with the minimum number of nonrequired elements. In this case, we can decrease the number of colors by increasing the value α while the number of additionally required elements decrease only a little bit. The new algorithm would look like the following pseudocode,

So if we choose $\alpha = 0$, we would have the same algorithm as before. As in Figure 3.12, the number of additionally required elements would increase. The number of colors is almost in the same order as in Figure 3.13. It can be compared to the computation on the same configuration but with different $\alpha = 6$. The number of colors would increase as in Figure 3.15 while the number of additionally required elements would not decrease a lot as

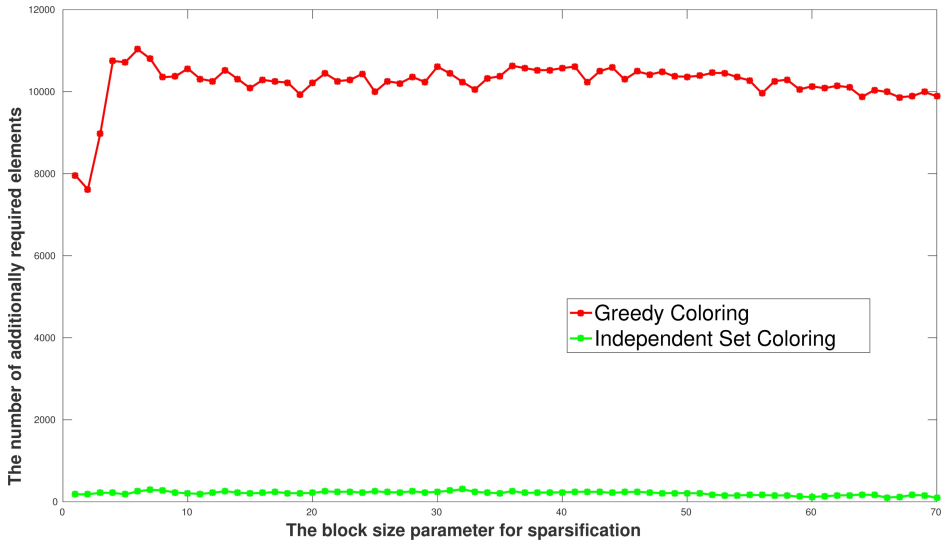


Figure 3.11: The comparison of an algorithm which color the independent sets first and the greedy algorithm with respect to the number of additionally required elements.

in Figure 3.15.

Here, we compute the new heuristic on the other matrix *nos3* and the results for $\alpha = 10$ are shown in Figure 3.16 and Figure 3.17 and for $\alpha = 1$ are shown in Figure 3.18 and in Figure 3.19. The new computations shows almost the same results. However, the only difference is that all the results change dramatically after the size of block 40. In general, an observation can be summarized that the results are better than for the smaller size of blocks specially when the matrix is small.

The adaption of this new algorithm to the bipartite graph mode is straightforward. Only two parts of algorithm needs this adaption. The part which the incidency is checked in which a distance-2 neighbourness should be checked instead of normal neighbourness. Also, the part that we search for the indpendent set containing a given vertex should be computed again by a distance-2 neighbourness. Algorithm 3.5

3.2.2 Star bicoloring

The classical approach toward star bicoloring is first introduced in [3] as star bicoloring scheme. This algorithm is implemented and evaluated in

```
1  function d2_color_nreq(G_b,V_c,α)
2    for v ∈ V_c with Φ(v) = 0
3      forbiddenColors[0] = v
4      if ∃n ∈ N(v) : (v,n) ∈ E_i
5        for n ∈ N(v)
6          if Φ(n) ≠ 0
7            forbiddenColors[Φ(n)] = v
8
9      Φ(v) = min{a > 0 : forbiddenColor[a] ≠ v
10     I_v = {u ∈ V_c : u ∉ N_2(v)}
11     nreq_v = {n ∈ N(v) : (n,v) ∉ E_i}
12     for i ∈ I_v
13       nreq_i = {n ∈ N(i) : (n,i) ∉ E_i}
14       req_i = {n ∈ N(i) : (n,i) ∈ E_i}
15       mapVtoNreq = mapVtoNreq ∪ (i, count(nreq_i − nreq_v))
16       mapVtoreq = map ∪ (i, count(req_i))
17     maxs = {a ∈ keys(map) : map[a] = max(values(mapVtoNreq))}
18     mins = {a ∈ keys(map) : map[a] = min(values(mapVtoNreq))}
19     minReq = {a ∈ maxs : mapVtoreq[a] = min(values(mapVtoreq))}
20     Φ(minReq) = Φ(v)
21     for i ∈ {0,1,...,α}
22       Φ(mins[i]) = Φ(v)
```

Algorithm 3.4: New coloring heuristc with a controller to balance the number of colors and the number of additionally required elements.

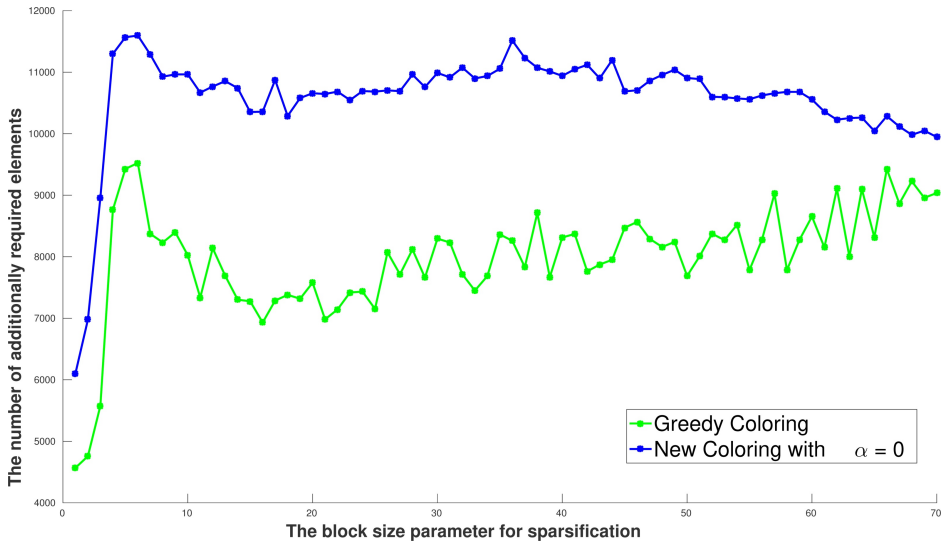


Figure 3.12: The number of additionally required elements computed by the new heuristics compared with the greedy algorithm. The computation carried out on the matrix *ex33* and value of α is set to 0.

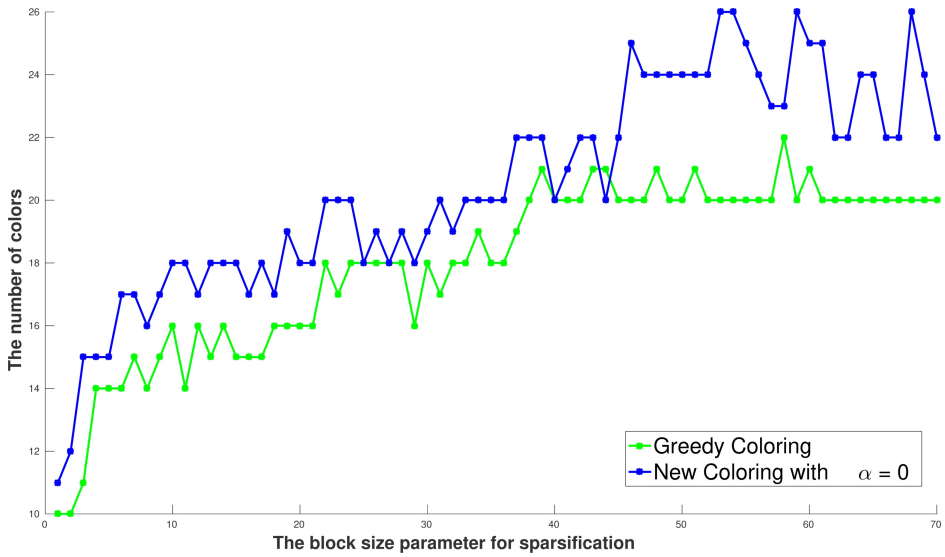


Figure 3.13: The number of colors computed by the new heuristics compared with the greedy algorithm. The computation carried out on the matrix *ex33* and value of α is set to 0.

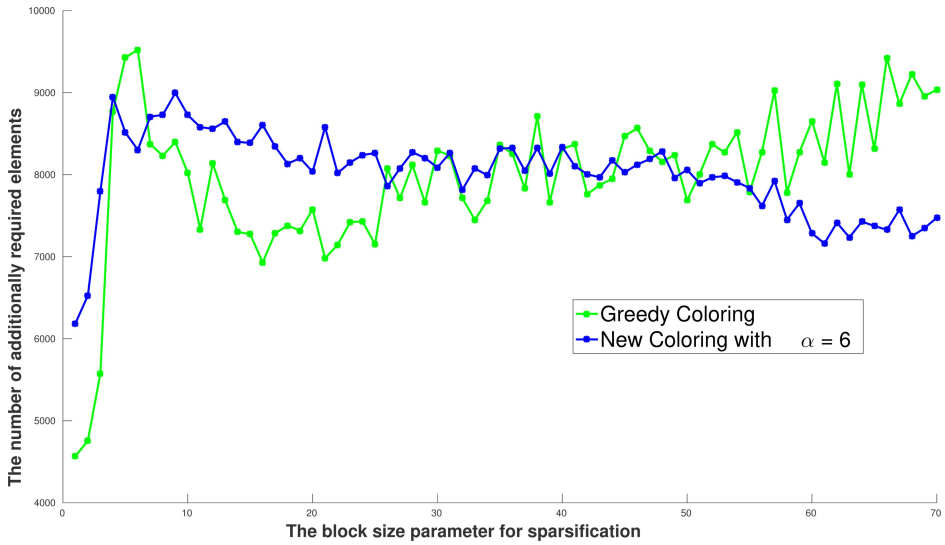


Figure 3.14: The number of additionally required elements computed by the new heuristics compared with the greedy algorithm. The computation carried out on the matrix *ex33* and value of α is set to 6.

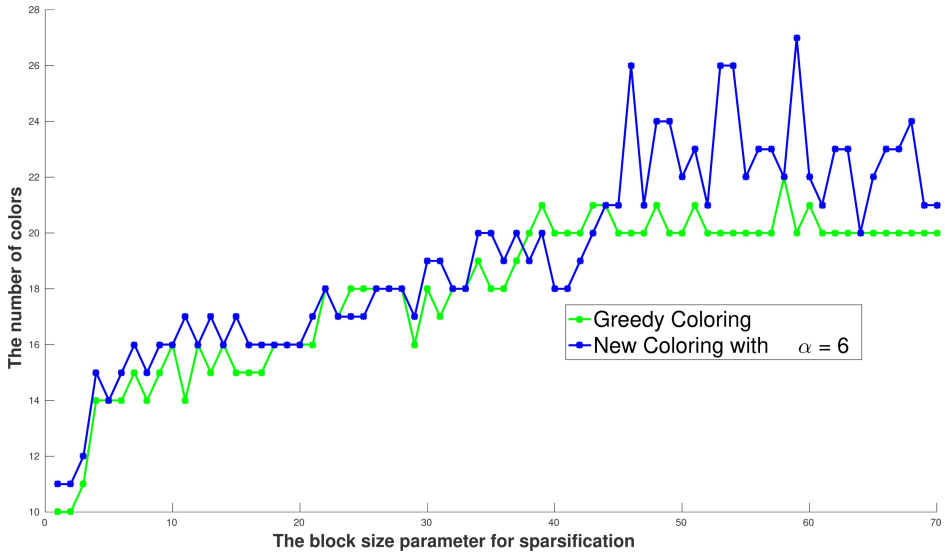


Figure 3.15: The number of colors computed by the new heuristics compared with the greedy algorithm. The computation carried out on the matrix *ex33* and value of α is set to 6.

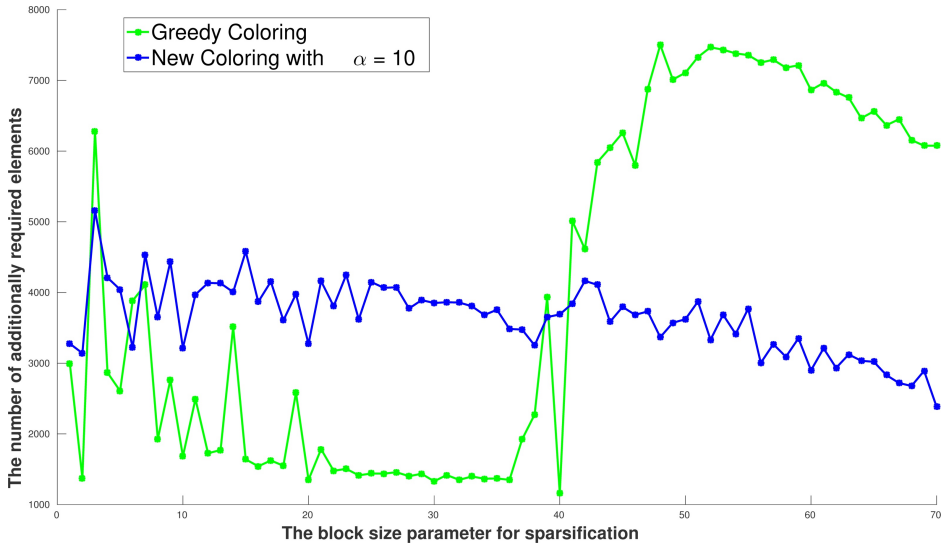


Figure 3.16: The number of additionally required elements computed by the new heuristics compared with the greedy algorithm. The computation carried out on the matrix *nos3* and value of α is set to 10.

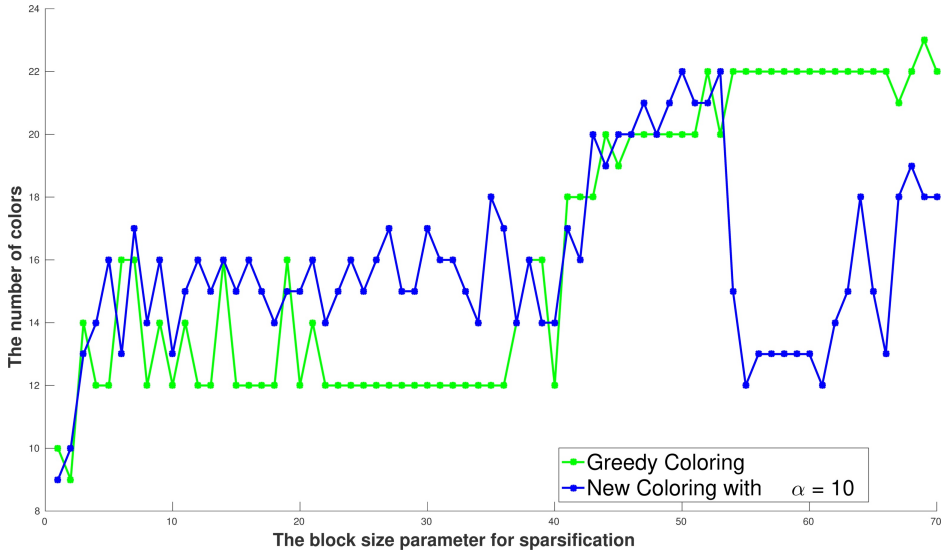


Figure 3.17: The number of additionally required elements computed by the new heuristics compared with the greedy algorithm. The computation carried out on the matrix *nos3* and value of α is set to 10.

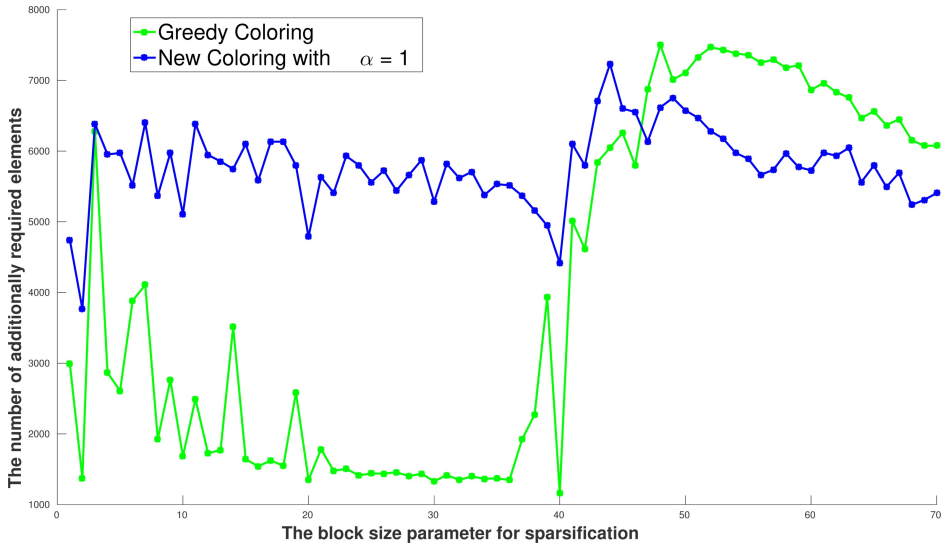


Figure 3.18: The number of additionally required elements computed by the new heuristics compared with the greedy algorithm. The computation carried out on the matrix *nos3* and value of α is set to 1.

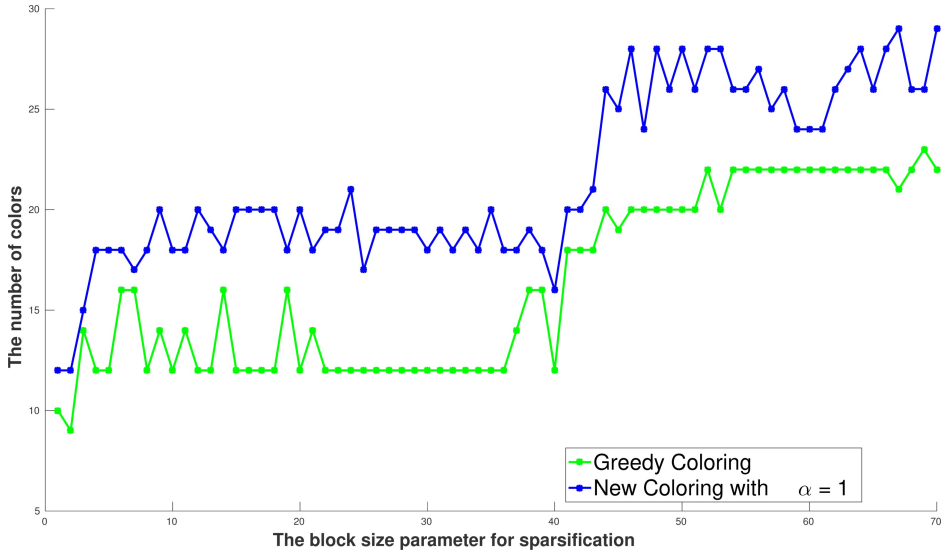


Figure 3.19: The number of additionally required elements computed by the new heuristics compared with the greedy algorithm. The computation carried out on the matrix *nos3* and value of α is set to 1.

```
1  function d2_color_nreq(G_b,α)
2    for v ∈ V_c with Φ(v) = 0
3      forbiddenColors[0] = v
4      if ∃n ∈ N_2(v) : (v,n) ∈ E_i
5        for n ∈ N_2(v)
6          if Φ(n) ≠ 0
7            forbiddenColors[Φ(n)] = v
8
9      Φ(v) = min{a > 0 : forbiddenColor[a] ≠ v
10     I_v = {u ∈ V_c : u ∉ N_2(v)}
11     nreq_v = {n ∈ N_2(v) : (n,v) ∉ E_i}
12     for i ∈ I_v
13       nreq_i = {n ∈ N_2(i) : (n,i) ∉ E_i}
14       req_i = {n ∈ N_2(i) : (n,i) ∈ E_i}
15       mapVtoNreq = mapVtoNreq ∪ (i, count(nreq_i − nreq_v))
16       mapVtoreq = map ∪ (i, count(req_i))
17     maxs = {a ∈ keys(map) : map[a] = max(values(mapVtoNreq))}
18     mins = {a ∈ keys(map) : map[a] = min(values(mapVtoNreq))}
19     minReq = {a ∈ maxs : mapVtoreq[a] = min(values(mapVtoreq))}
20     Φ(minReq) = Φ(v)
21     for i ∈ {0,1,...,α}
22       Φ(mins[i]) = Φ(v)
```

Algorithm 3.5: New coloring heuristc rewritten in the bipartite graph model

Lülfesmann [31]. This algorithm performs convincing since the steps of the algorithms is near to the definition of the coloring. Complete direct solver bicoloring and Row/Column Fill Bicoloring [32] are the other algorithms which are introduced for bicoloring. As Calotoiu [33] dicusses theses algorithms do not perform better than the classic star bicoloring scheme. Calotoiu [33] introduces the integrated star bicoloring and total ordering star bicoloring which performs better than star bicoloring scheme in some matrices and not much worse in some other matrices.

As same as the last section, we search for a modified version of star bicoloring which increases the number of additionally required elements without a high increase the number of colors. The idea from the new heuristic can be applied here. However it can survive less additionally required elements since a survived nonrequired elements from the column compression it is not necessary a survived nonrequired elemtns form the row compression.

We consider the following implemenetation of star bicoloring based on Algorithm 3.5 from Lülfesmann [9]. Here, the notation $G[S]$ means the graph induces on the edge set S.

```
1  forbiddenColors = [0, 0, ..., 0]
2  Φ = [−1, ..., −1]
3  v = −1
4  E'_R = E_R
5  while E'_R is not empty
6    if(Δ(V_r, G[E'_R]) > ρΔ(V_c, G[E'_R]) then
7      v = v_r ∈ V_r with maximum degree in G[E_R^{-1}]
8    else
9      v = v_c ∈ V_c with maximum degree in G[E_R^{-1}]
10   E'_R = E'_R − (v, w) ∈ E'_R : w ∈ N_1(v, G[E'_R])
11   for all w ∈ N_1(v, G)
12     if Φ(w) ≤ 0
13       for all x ∈ N_1(w, G) with Φ(w) > 0
14         if (v, w) ∈ E_R or (w, x) ∈ E_R
15           forbiddenColors[Φ(x)] = v
16     else
17       for all x ∈ N_1(w, G[E_R]) with Φ(w) > 0
18         for all y ∈ N_1(x, G) with Φ(y) > 0 and y ≠ w
19           if Φ(w) = Φ(y)
20             forbiddenColors[Φ(x)] = v
21   Φ(v) = minj > 0 : forbiddenColors[j] ≠ v
22   for all v_c ∈ V_c with Φ(v_c) > 0
23     Φ(v_c) = Φ(v_c) + maxΦ(v_r) : v_r ∈ V_r
24   for all v ∈ V_r ∪ V_c with Φ(v) ≠ −1
```

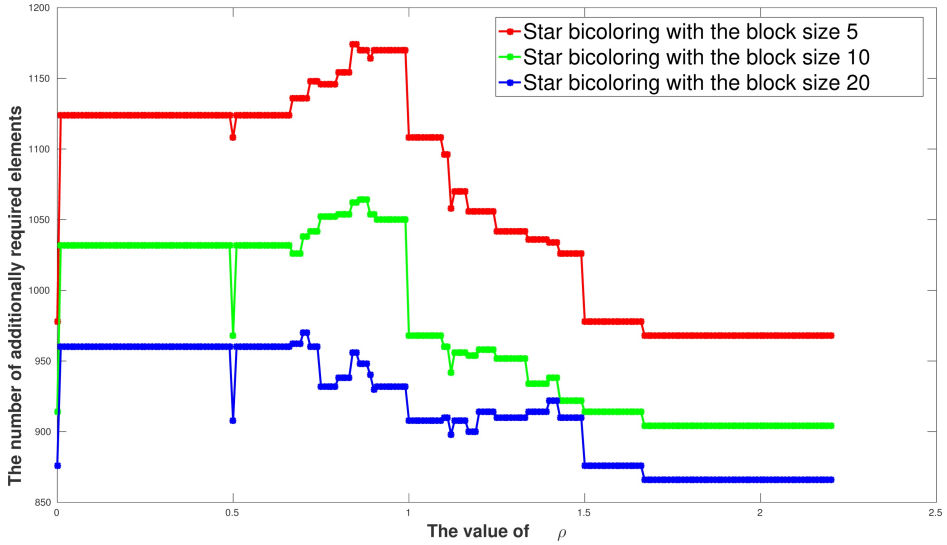


Figure 3.20: influence of ρ

25 $\Phi(v) = 0$

This algorithm process a vertex either from column vertices or row vertices with maximum degree in each step. We first did some computations to find the influence of ρ on the number of additioanlly required elements. The value of ρ is a weightening factor which is a balance between columns or row vertices. A higher value of ρ makes the compression mostly in the column vertices and vice versa. There are some discussion on how to choose the value of ρ in [3]. Also, Lülfesmann [9, 31] did som computation for some specific ρ. However, the main goal in these previous literature is to minimize the number of colors. As the Figure 3.20 and Figure 3.21, the value of ρ has definitely a direct influence on the number of additionally required elements. The interesting observation is that a tiny change on the value of ρ can dramatically change the results. For example, changing the value of ρ from 0.3 to 0.4 in Figure 3.21 would result in a change of 10000 in the number of additionally required elements.

As we said, the first part of algorithms chooses the next vertex which should be processed from the maximum degree vertices in a graph induces on the required edgs. We change this part of algorithm to process a specific algorithm with maximum degree and maximum nonrequired elements instead of an arbitrary vertex. After the first step, this processes can be repeated or

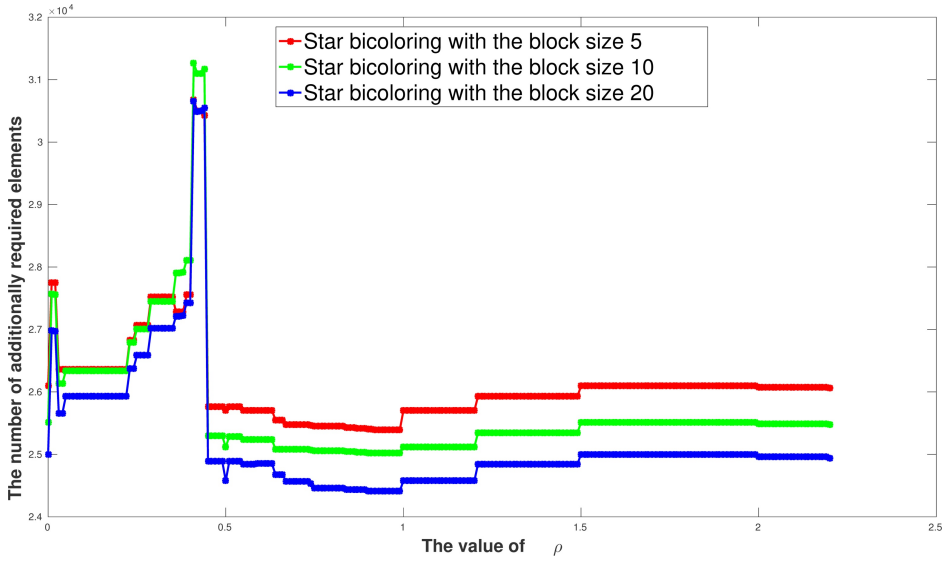


Figure 3.21: influence of ρ

we can process the vertices with minimum number of nonrequired elements.

3.2.3 Parallelization

As we sawe in Section 2.3, different steps should be followed to compute the additionally required elements. Here, we look at each of these steps and examine how they can be parallelized by OpenMP if it can be done efficiently.

First, we look at the coloring algorithm. There are many literature for a parallelized coloring algorithm. For example, Çatalyürek [34] itroduced an OpenMP parallelized algorithm which comptues the same number of colors as the serial version. However, there are two points in each step of the algorithm in which two threads need to be synchronized. However, we focus on another algorithm from Rokos et al [35] which presents an algorithm in which only a point of synchronization is needed. The number of colors also remains near to the number of colors in the serial version. A bipartite version of the algorithm can be easily adapted as in Algorithm 3.6. This algorithm first colors the vertices greedily and then tries to correct the false colorings which can happend. This idea of parallelization can be applied to our new coloring heuristics. It means we compute the coloring like before by a parallelized loop. Then, we correct the coloring. Algorithm 3.7 shows the parallelized

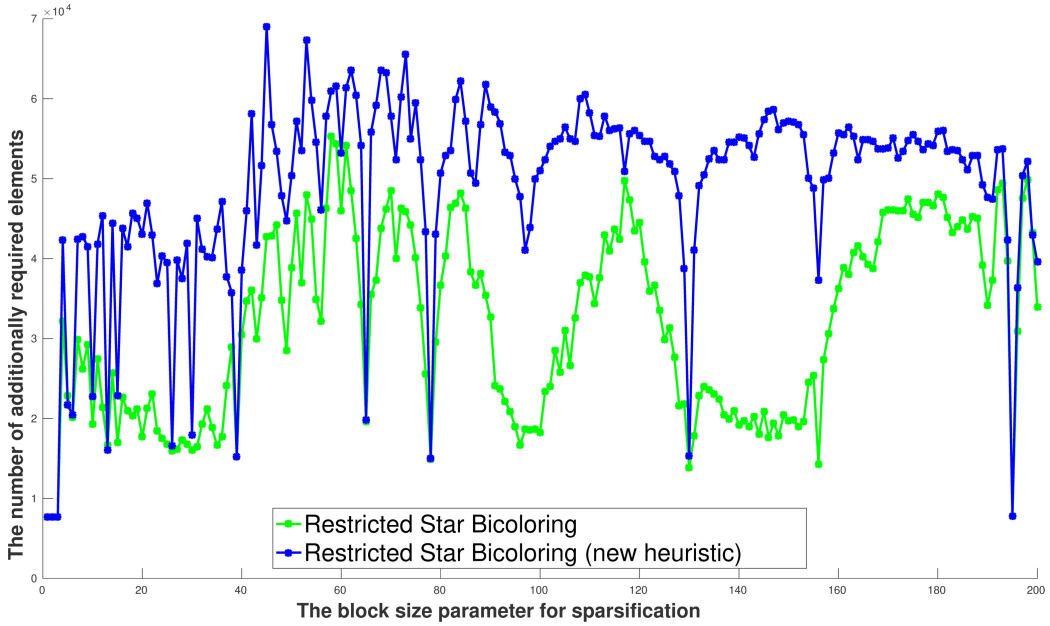


Figure 3.22: The number of additionally required elements computed with the new star bicoloring compared with the older implementation. The asymmetric matrix *crystm*01 is used here.

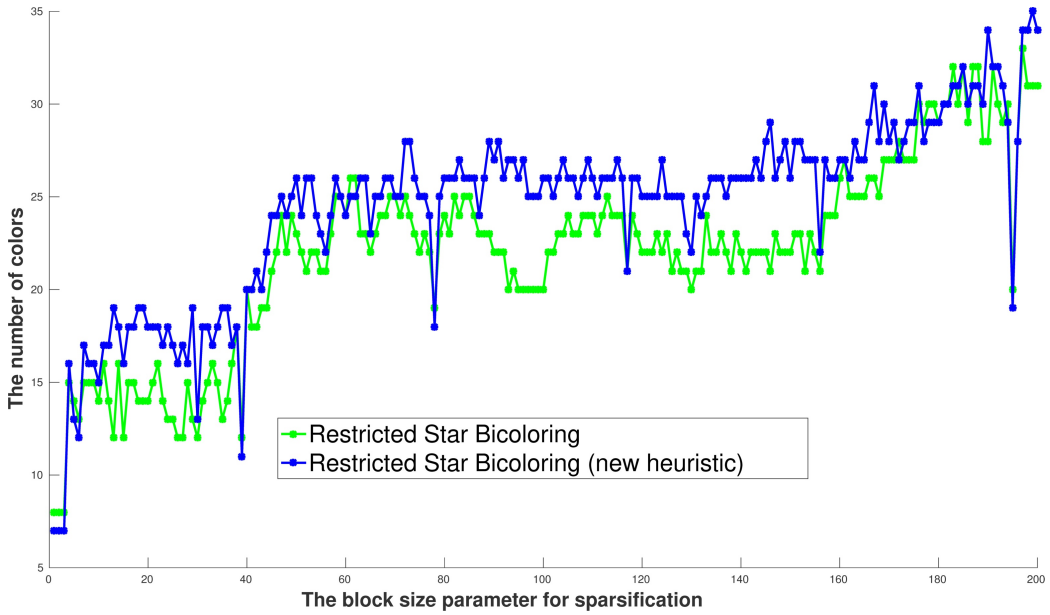


Figure 3.23: The number of additionally required elements computed with the new star bicoloring compared with the older implementation. The asymmetric matrix *crystm*01 is used here.

```
function d2_color_omp($G$,$V$)
  #progrma omp parallel for
  for $v\in V$
    $forbiddenColors[\Phi(n)] = v$
    $\Phi(v) = min \{ a>0:forbiddenColor[a]\neq v$
  #pragma omp barrier
  $U_0 = V$
  $i = 1$
  while $U_{i-1}\neq \emptyset$
    $L=\emptyset$
    #pragma omp parallel for
    for $v\in U_{i-1}$
      if $\exists u\in N_2(v), u>v: \Phi(u) = \Phi(v)
        $forbiddenColors[\Phi(n)] = v$
        $\Phi(v) = min \{ a>0:forbiddenColor[a]\neq v$
    #pragma omp barrier
    $U_i = L$
    $i = i+1$
```

Algorithm 3.6: A OpenMP parallelized version of greedy algorithm adapted for the bipartite graph.

version of Algorithm 3.5.

We color the bipartite graph intrepeted from the matrix *Cavity16* from the sparse matrix collection of University of Florida. The timing are all from the computation carried on an intel Core i5 with 8 Gb RAM. Table 3.24 shows the results of this computation. Here, we change the number of threads from 1 to 10 shown in the first column. The second colum shows the computation time in milliseconds. The third column is also the number of colors which changes based on the number of colors. Table 3.24(Left) shows the results of computation of Algorithm 3.6 and Table 3.24(Right) for Algorithm 3.7. Also Figure 3.25 (Left) and Figure 3.25 (Right) visualize the speedup based on the number of threads.

This idea can be applied to the new heuristic for star bicoloring which is a potential future work.

Apart from the parallelization of coloring, the block diagonal sparsification makes a suitable matrix for parallelization. It is well known the fill-ins are generated only in the nonzero blocks in such matrices. This observation gives a direct idea of parallelization in which each process works on each block. We do not go into details since it is already discussed in previous

```
1  function d2_color_nreq(G_b,α)
2    #progrma omp parallel for
3    for v ∈ V_c with Φ(v) = 0
4      forbiddenColors[0] = v
5      if ∃n ∈ N_2(v) : (v,n) ∈ E_i
6        for n ∈ N_2(v)
7          if Φ(n) ≠ 0
8            forbiddenColors[Φ(n)] = v
9
10     Φ(v) = min{a > 0 : forbiddenColor[a] ≠ v
11     I_v = {u ∈ V_c : u ∉ N_2(v)}
12     nreq_v = {n ∈ N_2(v) : (n,v) ∉ E_i}
13     for i ∈ I_v
14       nreq_i = {n ∈ N_2(i) : (n,i) ∉ E_i}
15       req_i = {n ∈ N_2(i) : (n,i) ∈ E_i}
16       mapVtoNreq = mapVtoNreq ∪ (i, count(nreq_i − nreq_v))
17       mapVtoreq = map ∪ (i, count(req_i))
18     maxs = {a ∈ keys(map) : map[a] = max(values(mapVtoNreq))}
19     mins = {a ∈ keys(map) : map[a] = min(values(mapVtoNreq))}
20     minReq = {a ∈ maxs : mapVtoreq[a] = min(values(mapVtoreq))}
21     Φ(minReq) = Φ(v)
22     for i ∈ {0,1,...,α}
23       Φ(mins[i]) = Φ(v)
24
25   #pragma omp barrier
26   U_0 = V
27   i = 1
28   while U_{i−1} ≠ ∅
29     L = ∅
30     #pragma omp parallel for
31     for v ∈ U_{i−1}
32       if ∃u ∈ N_2(v), u > v : Φ(u) = Φ(v)
33         forbiddenColors[Φ(n)] = v
34         Φ(v) = min{a > 0 : forbiddenColor[a] ≠ v
35     #pragma omp barrier
36     U_i = L
37     i = i + 1
```

Algorithm 3.7: New coloring heuristc in the bipartite graph model parallelized by OpenMP

Threads	Time	Colors
1	42.8745	47
2	33.9665	47
3	25.2741	48
4	20.6863	48
5	21.4943	47
6	20.1796	50
7	17.9640	49
8	16.1068	52
9	15.4174	47
10	16.1545	47

Threads	Time	Colors
1	96.795	48
2	75.744	47
3	55.605	49
4	49.335	49
5	49.360	47
6	49.365	51
7	45.342	47
8	43.254	51
9	40.742	48
10	39.416	47

Figure 3.24: (Left) The results of computation of greedy coloring parallelized by OpenMP. (Right) The results of computation of new heuristics parallelized by OpenMP

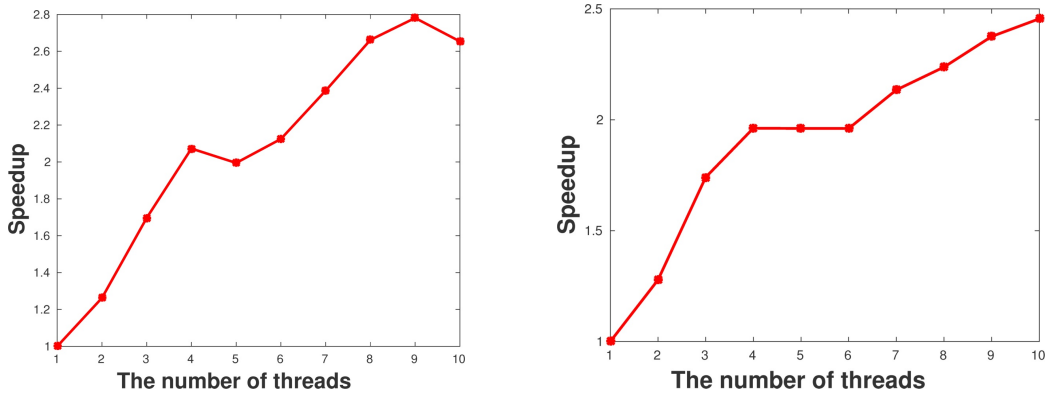


Figure 3.25: (Left) Speedup for the computation of greedy coloring in bipartite graph model parallelized by OpenMP. (Right)Speedup for the computation of new coloring heurtistic in bipartite graph model parallelized by OpenMP.

literature like [36]. However, it does not mean that the same idea can be applied in computation of additionally required elements since those elements are outside the blocks.

3.3 PreCol 1.0

Here, we explain what we have done to ease the usage of the software either for developers or for end users. We have restructured the implementation to achieve these goals.

We employed concepts from the object-oriented programming to empower the developers to implement new extensions without going into details of core implementation. So, two main ingredients coloring and orderings can be implemented now only by deriving an interface. For example, a new ordering can be added as easy as the following code.

```
1 class LFO : public Ordering {
2     bool order(const Graph &G_b, vector<unsigned int> &ord,
          bool restricted) {
3         ...
4     }
5 };
```

The developer needs to implement this new class in a only-header fashion [37], since the goal is to write an extension with a few code. So, the developer should move the new header file to the corresponding directory which is the ordering directory for this new ordering and *algs* directory for coloring algorithms. Now, building the software would bring this new ordering into the software execution.

Using the standard library of C++ as well as the concept of functional programming in new C++ release [38], we provide different functions which can be used by developer to work on graphs. For example, the iteration of vertices or edges can be easy as follows,

```
1 for_each_v(G, f);
2 for_each_v(G, [&](Ver v) {...});
3 for_each_e(G, f);
4 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, respectively. Also, the other syntax is the lambda function from the new C++ functional programming to implement a function in place.

Following a unique solution, we implement all the possible part of algorithms with the use of standard library of C++ which also improves the readability. This strategy reduces the amount of the code dramatically and make the code more readable. Also, the standard algorithms would be automatically parallel in the next release of C++ (C++17) next year [39].

3.3.1 Restructuring.

In the previous version, the coloring, potentially required elements, and the additionally required elements were computed in the functions written in C++. The other parts of computation were done in Matlab. This means that the code in Matlab needs to call the MEX interface for three times at least. In this way, it was necessary to do the conversion from the Matlab sparse matrix to graph (or matrix) format in C++ and converting back from C++ to Matlab. This is illustrated in Figure 3.26 (Left). This results in the lack of efficiency. On the other hand, this mix of programming languages reduces the accuracy and makes the versioning difficult.

We improved the design of the software to resolve these problems. So, all the parts of algorithm are computed in *C++* now. Figure 3.26 (Left) visualizes this new implementation. All the parts can be computed in C++. However, the part of iterative solver part, can be computed also in MATLAB, since the MATLAB functions in this area are in general more convincing. For this goal, we implemented the preconditioning and and the sparsification in C++. As you can see in Figure 3.26 (Left), we also implemented two interfaces in MATLAB and JAVA languages which we will explain in Section 3.3.2.

To use the software, the user can use a command-line command in addition to the interfaces in JAVA and MATLAB. So, the user needs to specify different options for coloring algorithm, orderings, the block size, and so. These options can be entered directly in the terminal together with command or can be written in a so called config file which is imported by the software. The format of config file is as follows,

```
1 alg: PartialD2Coloring
2 col_ord: Min
3 ilu_ord: LFO
4 type: BlockDiagonal
5 blk: 10
```

Different parameters possible in config file is as follows,

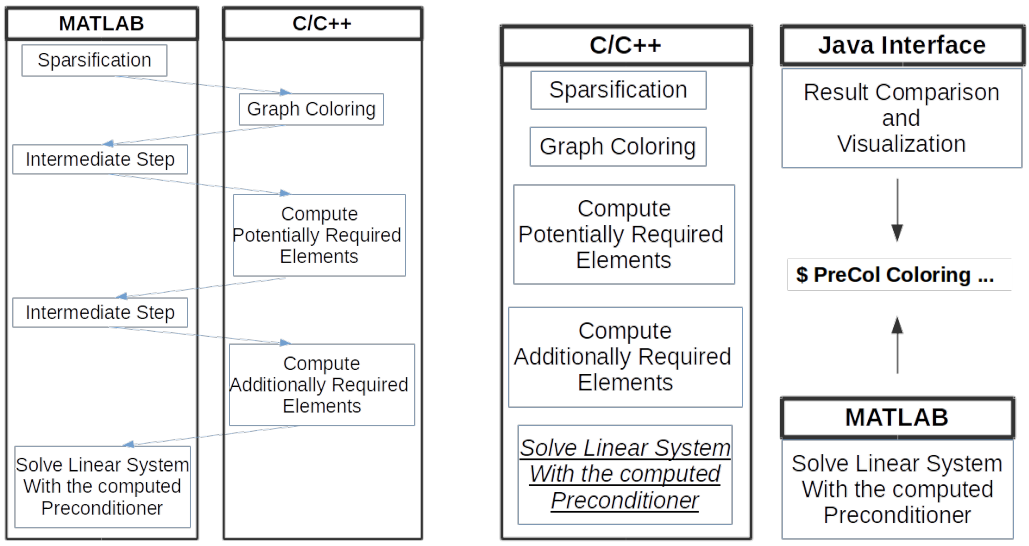


Figure 3.26: There were many communications between MATLAB and C/C++ version. The communication is done based on the MEX interface. A conversion from matrix to graph is computed each time that the MEX interface is used. (Left) A new implementation reduces a lot of communication. It brings together different parts of algorithm in a package. Two interfaces written in MATLAB and Java call this package as a local program.

- alg:
- col_ord:
- type:
- blk:

We also generated a new documentation as a website which is available under the website: http://csc.c3e.de/precol/html. This website is generated automatically by Doxygen [40]. We implemented some new HTML/CSS template for doxygen to include extra texts and documentation in the website.

3.3.2 JAVA and MATLAB interface

JAVA interface. We extend our JAVA software GraphTea [41, 42, 43, 44, 45, 46, 47] to have a set of reports for graph coloring and preconditioning which call the program *PreCol*. GraphTea is a graph editing framework designed specifically to compute and visualize different parameters of graphs interactively. Figure 3.27 shows a snapshot of the main graph window together with two additional windows that give more details on the solution of different graph problems. These separate windows providing additional information are called "reports."

A report can be computed in GraphTea in two different way: a single report or an incremental report. A single report means to compute some parameters on graph and look at the results in a textual way. However, an incremental report happens when we have at least two parameter for computation. So, we would change one of the parameter in some range and would generate a table.

The main goal of developing GraphTea was to help the researcher through their research. The following abilities of GraphTea could help the research in different dimensions:

- Generate different class of graphs and compute the parameters on them.
- Get reports on the graph beside the parameter and find the connections.
- Compute the operations on graphs and influence of these operations on parameters of the graph.

If we divide the researcher's works follows,

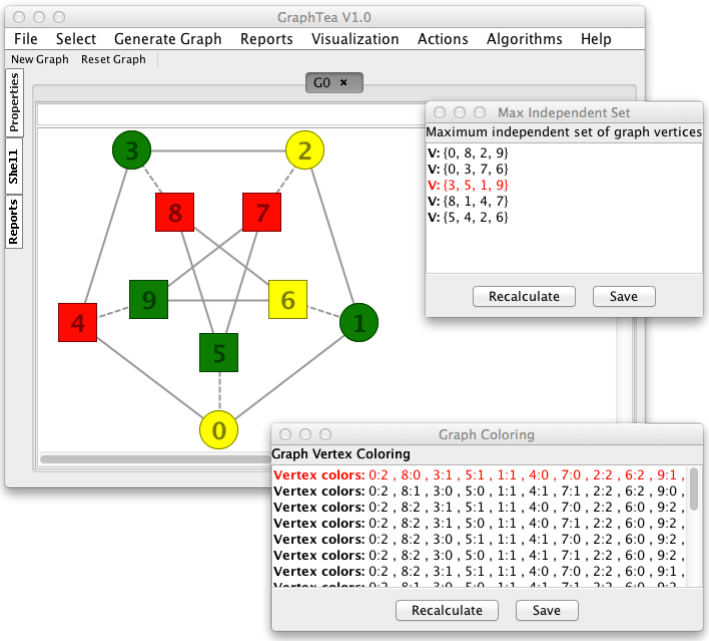


Figure 3.27: An overview of GraphTea: A graph drawing window and two floating dialogs with reports on a given graph.

1. Guess a hypothesis like a bound for coloring number
2. Evaluate this bound on different graphs
3. Prove the hypothesis

The second step, i.e. evaluation, is always an important step since many first errors and improvements could be found. However, the researcher can only do this evaluation for small examples if they do not use computer tools. Different aspects of GraphTea can be used to overcome this evaluation. The researchers can generate any graph up to any computationally-reasonable size (i.e. commutable in a reasonable time) and compute the parameter. This process of so-called conjecture checking on different graphs could often lead to a better guess.

Beside the automatic generation of graphs up to any size, we added the ability of loading different sparse matrix as graphs to GraphTea. A new data structure for the sparse matrix, called *SpMat*, is designed to handle the large sparse matrices. This data structure is basically an array. This array has

the size of number of rows. Each cell of this array points to an array which is the corresponding columns of nonzeros in this row.

MATLAB interface. zero or one mohem ... discussion about the sparsification in MATLAB and C/C++ staring from zero or one ???

Chapter 4

Interactive Educational Module

EXPLAIN is an extensible collection of educational modules for classroom use. It is currently not designed for self-study because the connection between the scientific computing problem and the corresponding graph problem is not available in EXPLAIN. The idea is that the teacher will explain this connection as well as the use of the module in classroom. EXPLAIN has so far two major releases EXPLAIN 1.0 and EXPLAIN 2.0. In implementation details we explained how they differ and improved from an original version.

We implemented different modules for EXPLAIN. Here we explain these modules and how they can be used. The implementation details comes later in 4.7.

4.1 Column Compression

In [48, 49], we presented an educational module in EXPLAIN to visualize the coloring algorithm for the column compression interactively. The idea is summarized as follows.

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 a predefined list, $\{1 = \text{green}, 2 = \text{turquoise}, 3 = \text{orange}, 4 = \text{violet}, 5 = \text{red}, 6 = \text{yellow}, ...\}$, such that it differs from the colors of those neighbors that are already colored. Recall that a group of columns corresponds to a set of vertices in the graph with the same

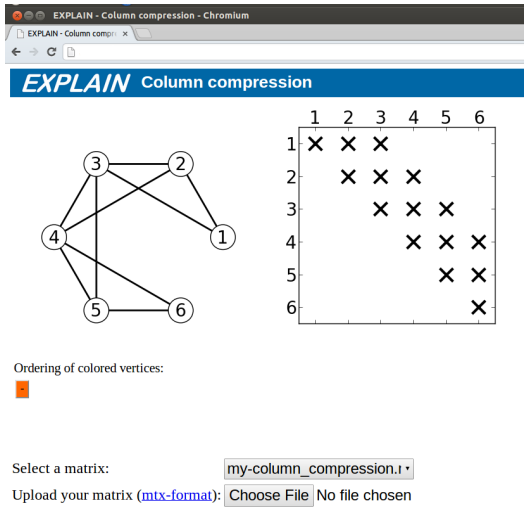
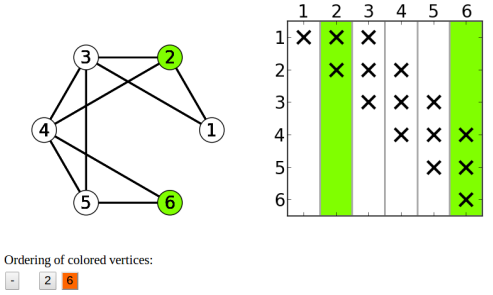


Figure 4.1: The initial layout of EXPLAIN for some given column compression problem. Matrix and graph are visualized side by side to recognize the underlying connection between these two equivalent representations. Nonzero elements are indicated by the symbol $\times$.

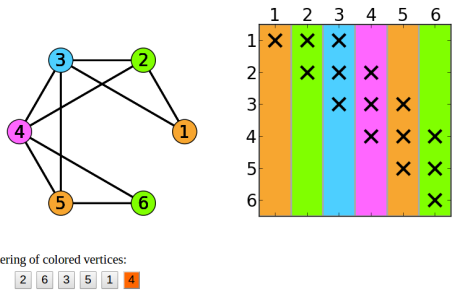
color. To indicate this, we do not color only the vertices in the graph but also the corresponding columns in the matrix.

Figure 4.1 shows a screenshot of the column compression module. The nonzero elements of the matrix are denoted by the symbol $\times$ in the matrix view; the corresponding column intersection graph is given immediately next to the matrix. 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.

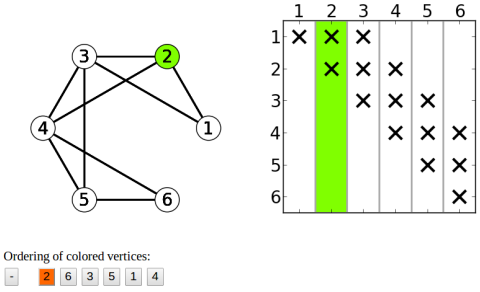
Suppose a vertex is selected in the first step. This vertex is then colored using the first color of the predefined list. Continuing this process of vertex selection, different colors are chosen and an ordered list of vertices is created which is already indicated in Figure 4.1 marked by "Ordering of colored vertices." In this figure the list is empty but it will become nonempty in later figures. Each button of this list is clickable, causing EXPLAIN to return back to that step of the algorithm. The process is continued until all



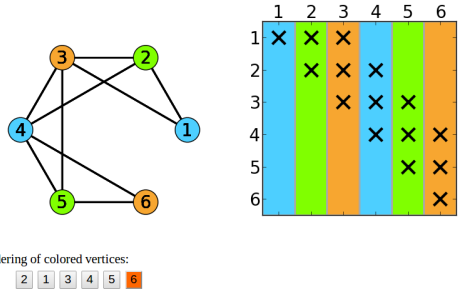
(a) Student selected v_2 and v_6 in that order.



(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 4.2(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 4.2: Display of various situations after interactively choosing vertices.

vertices are colored. The button labeled by the minus sign, will return to the first step keeping the whole process history.

Figure 4.2(a) shows a representation of 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 graph in which the student has already selected the two vertices v_2 and v_6. Figure 4.2(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,

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

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

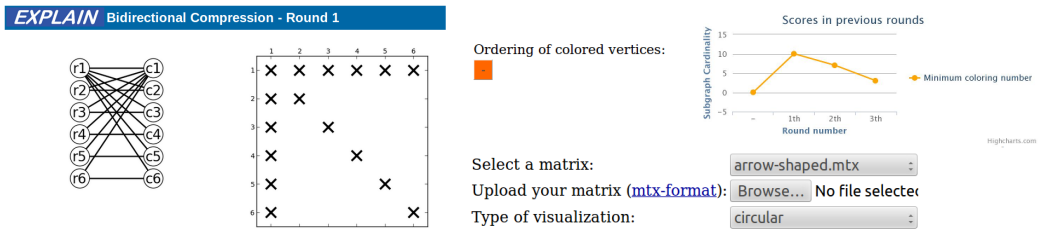
Since we want to provide the possibility to return back to some step of the algorithm, a history of the selection process is kept in the ordered vertex list. Now, suppose the student selects to return back to the step 1 where the vertex v_2 was selected, then the program returns back to that step of the algorithm. The resulting state is depicted in Figure 4.2(c). Notice that the program keeps the whole history and the student can click on any other vertex in the history.

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 4.2(d) leads to a reduction of one color compared to the first ordering given in Figure 4.2(b). Actually, this is the minimum number of colors needed to color this graph. The corresponding seed matrix is given by

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

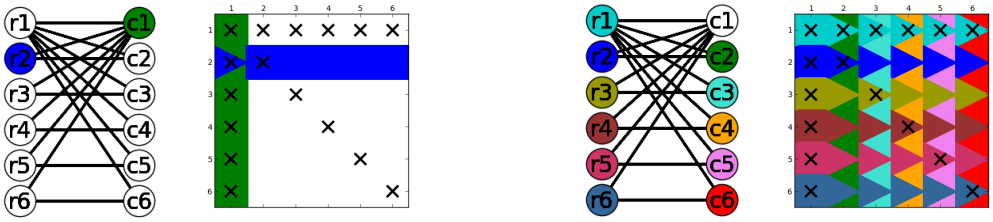
4.2 Bidirectional Compression

In our publication [50], we design and implement an interactive module to teach bidirectional compression and its connection to star bicoloring. Figure 4.3 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



(a) (b)

Figure 4.3: 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.

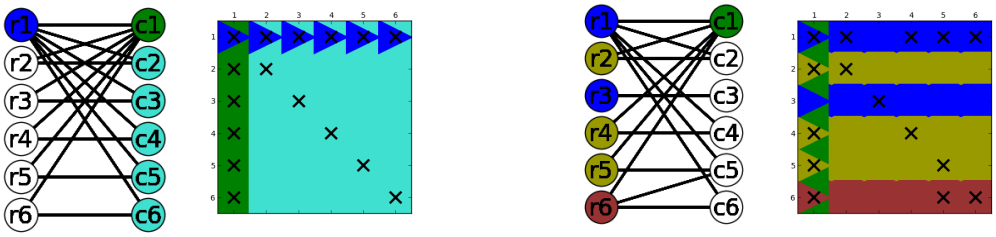


(a) (b)

Figure 4.4: The graph and the nonzero pattern (a) taken from Fig. 4.3 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.

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 ith row of the matrix. Likewise, a vertex on the right part of the graph labeled c_i corresponds to the ith column of the matrix. Different matrices can be selected from a predefined list or can be uploaded to the server using the menu depicted in Fig. 4.3 (b). We stress that EXPLAIN is designed for small problem instances.

Using any web browser, the student can interactively solve Minimum



(a) (b)

Figure 4.5: A star bicoloring (a) of the problem instance from Fig. 4.3 also considered in Fig. 4.4. 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

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. 4.3 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 4.4 (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. In EXPLAIN, 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. 4.3 (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. 4.4 (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. 4.5 (a) illustrates an exact solution of Minimum Star Bicoloring for this problem instance using the minimal 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. 4.3 (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 4.3 (b). The idea behind this score diagram is to use elements of game design to motivate and increase the student's activity in the learning process. This way, the student can see how successful he or she was in reducing the number of non-neutral colors.

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. 4.5 (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.

4.3 Partial Jacobian Computation

The previous module of matrix compression is extended to support also 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 the edges, their colors change to red and the corresponding nonzero elements are added to required elements.

As soon as the student starts to click the vertices, the required elements are become fixed, i.e. no new required elements can be added. The other processes of coloring is completely like the previous module.

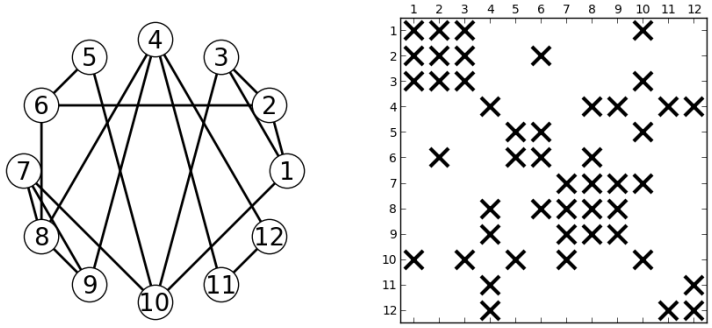


Figure 4.6: Two equivalent representations in terms of a graph (left) and a matrix (right).

4.4 Nested Dissection Ordering

4.4.1 Bisection

[51]

Though our primary aim is to design an educational model for illustrating the connection between scientific computing and graph theory, we also experience with ideas from gamification [52, 53]. 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 [54], 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 [55]. Programming assignment can also involve pieces of software that act as a player in an existing game. However, throughout the present paper, our focus of serious games is different. Rather than implementing a game, we are interested in situations where students learn by playing a game. Surprisingly, there are only a few publications addressing this aspect of gamification. An example is given in [56] where game-based learning is used to teach a course in data structures and algorithms. A collaborative game is described in [57] that aims at improving the teaching quality in a course on mathematical logic.

To engage the students more in the teaching process, we improved EXPLAIN such that the students get more feedback from the software. This

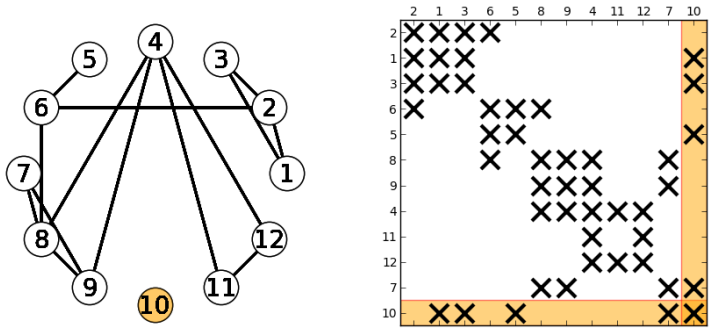


Figure 4.7: Graph and matrix view after selecting the vertex number 10. The decomposition into two blocks is still not shown as the graph is not yet decomposed into two disconnected components.

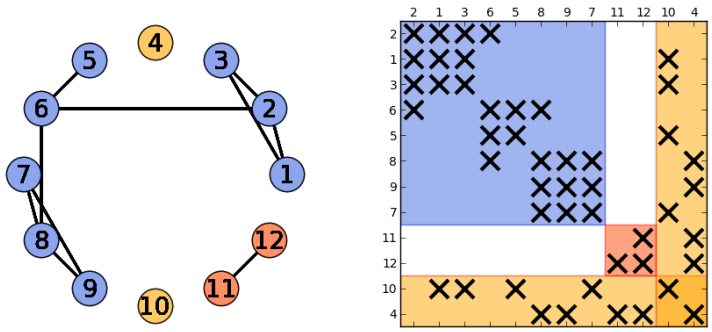


Figure 4.8: Graph and matrix view after selecting the vertices number 10 and then 4. The selection is not adequate as the sizes of blocks are not balanced.

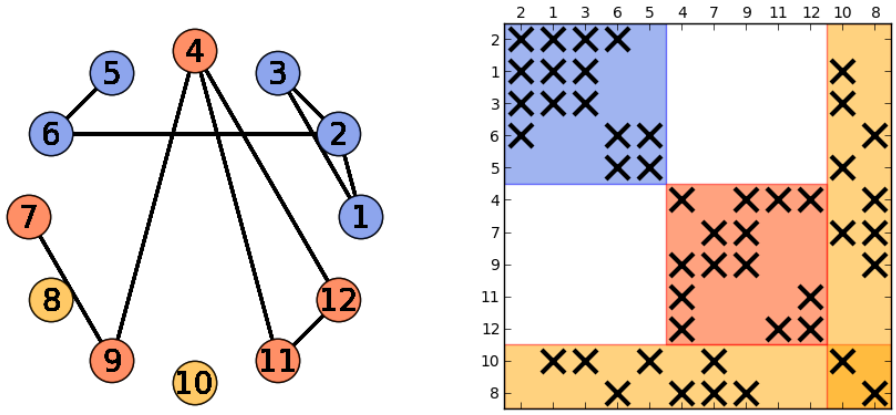


Figure 4.9: Graph and matrix view after selecting the vertices number 10 and then 8. The block sizes are balanced and the separator size is minimized.

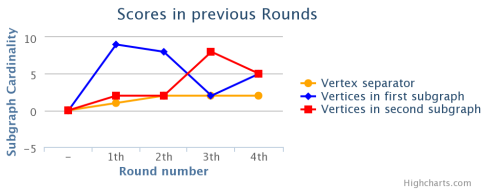


Figure 4.10: Score diagram resulting from four different rounds.

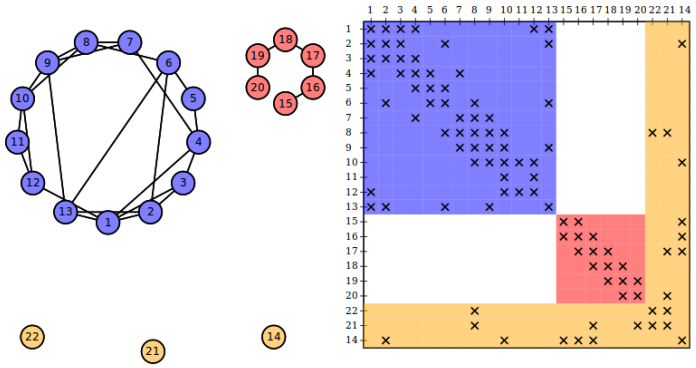


Figure 4.11: A selection results in an inefficient bisection of graph.

is done by gamification of the software by interpreting each solution to a problem instance as a round. The score diagram reporting the results of previous rounds also provides another feedback. The idea of gamification is used to solve a combinatorial minimization problem consisting of minimizing the size of the vertex separator while, at the same time, balancing the size of the remaining components. The consistent use of colors in the graph view, the matrix view, and in the score diagram makes it easier for the student to understand minimizing a serial bottleneck in the Cholesky factorization while balancing the computational load.

4.4.2 Recursive Disection

Based on the new features of EXPLAIN 2.0, we could improved the previous bisection to a recursive nested dissection algorithm. It contains the bisection itself. So, the student selects the vertex separator as before until the graph becomes disconnected. In this new version, the vertex separator will be shown on the bottom of the graph. Also the two remaining parts of graph are shown separately in two different circles at right and left. In Figure 4.11, the result of a selection is visualized which is inefficient since the parts of graphs are not balanced. Figure 4.12 also shows the results of an efficient selection.

Now, in contrast to the previous version, the student can click further on the vertices of each parts. This selection would trigger a recursive bisection of the matrix as well. Again, this selection goes forward until both parts of

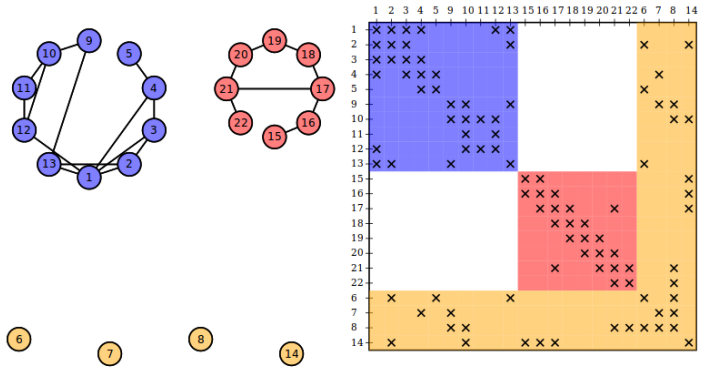


Figure 4.12: A selection results in an efficient bisection of graph.

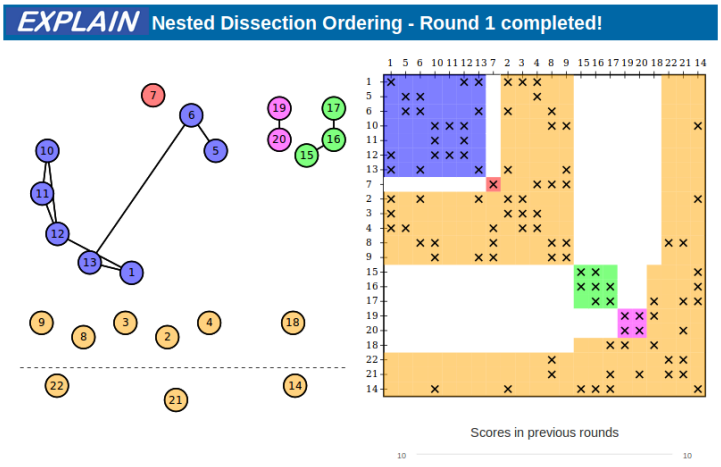


Figure 4.13: A complete nested dissection up to the order 2 which has an inefficient result.

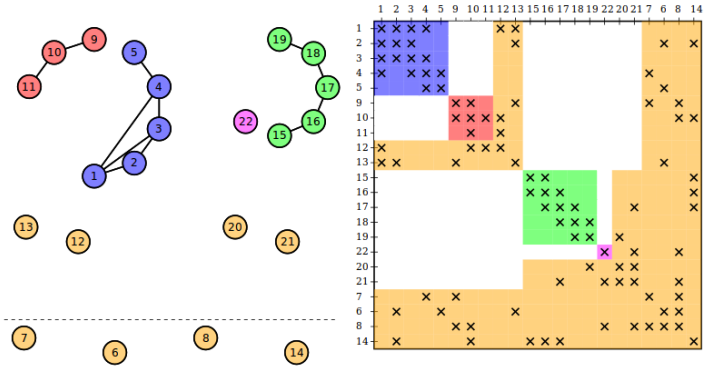


Figure 4.14: A complete nested dissection up to the order 2 which has an efficient result.

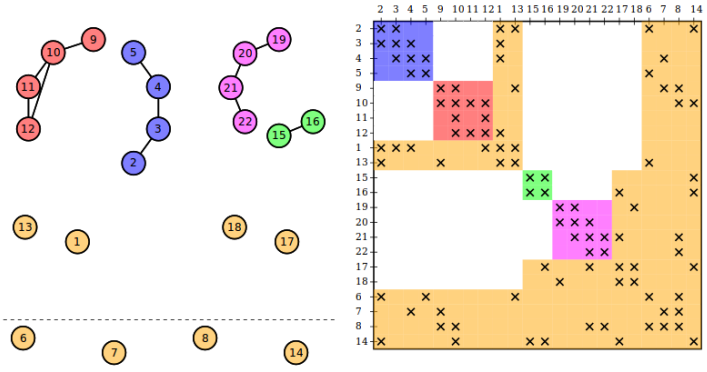


Figure 4.15: A complete nested dissection up to the order 2 which has an even more efficient result.

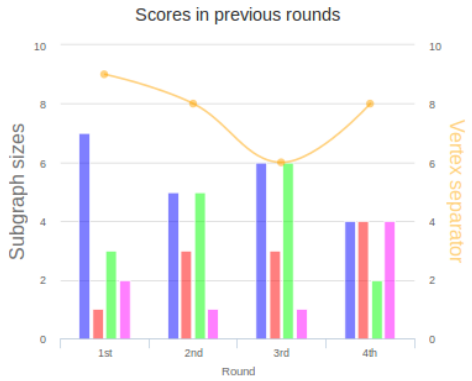


Figure 4.16:

the graphs become disconnected. The vertex separators would be moved to the bottom of the graph parts as well as the graph parts would be shown separately. Figure 4.13 and Figure 4.14 illustrated two inefficient and efficient results of nested dissection, respectively. Figure 4.15 shows how the results can get even better by a suitable order of selection.

The corresponding diagram of history of rounds are improved such that both the size of vertex separator as well as the size of four graph parts can be visualized. Figure 4.16 shows a possible selection history. As it can be seen the line chart shows the size history of the vertex separator and four bar chart grouped together shows the size history of the graph parts.

4.5 Parallel Matrix Vector Product

In [58], we extended EXPLAIN with a new module for the parallel sparse matrix-vector multiplication. So if we show the problem as follows,

$$\mathbf{y} = A \cdot \mathbf{x}$$

or by elements as follows,

$$\mathbf{y} \leftarrow A\mathbf{x} \tag{4.3}$$

where the N-dimensional vector $\mathbf{y}$ is the result of applying A to some given N-dimensional vector $\mathbf{x}$. Then, the ith entry of $\mathbf{y}$ is given by

$$y_i = \sum_{j \text{ with } a_{ij} \neq 0} a_{ij} \cdot x_j, \quad i = 1, 2, \ldots, N, \tag{4.4}$$

The question here is if data representing A, $\mathbf{x}$ and $\mathbf{y}$ are distributed to multiple processes, to what extent does this data distribution have an effect on the computation $\mathbf{y} \leftarrow A\mathbf{x}$? What are the advantages and disadvantages of a given data distribution? What are the criteria for evaluating the quality of a data distribution? How should data be distributed to the processes ideally?

To discuss such questions with undergraduates who are new to parallel computing we suggest to consider the following simple data distribution. The nonzero elements of A are distributed to processes by rows. More precisely, all nonzeros of a row are distributed to the same process. The vectors $\mathbf{x}$ and $\mathbf{y}$ are distributed consistently. That is, if a process stores the nonzeros of row i of A then it also stores the vector entries x_i and y_i. Given a fixed number of processes p, a data distribution may be formally expressed by a mapping called *partition*

$$P : I \to \{1, 2, \ldots, p\}$$

that decomposes the set of indices $I := \{1, 2, \ldots, N\}$ into p subsets I_1, I_2, $\ldots$, I_p such that

$$I = I_1 \cup I_2 \cup \cdots \cup I_p$$

with $I_i \cap I_j = \emptyset$ for $i \neq j$. That is, if $P(i) = k$ then the nonzeros of row i as well as x_i and y_i are stored on process k.

Since the nonzero a_{ij} is stored on process $P(i)$ and the vector entry x_j is stored on process $P(j)$, one can sort the terms of (4.4) according to those terms where both operands of the product $a_{ij} \cdot x_j$ are stored on the same process and those where these operands are stored on different processes:

$$y_i = \sum_{\substack{j \text{ with } a_{ij} \neq 0 \\ P(i) = P(j)}} a_{ij} \cdot x_j + \sum_{\substack{j \text{ with } a_{ij} \neq 0 \\ P(i) \neq P(j)}} a_{ij} \cdot x_j, \quad i = 1, 2, \ldots, N. \tag{4.5}$$

For the sake of simplicity, we assume that the product $a_{ij} \cdot x_j$ is computed by the process $P(i)$ that stores the result y_i to which this product contributes. By (4.5), the data distribution P has an effect on the amount of data that needs to be communicated between processes. It also determines which processes communicate with each other. Since, on today's computing systems, communication needs significantly more time than computation, it is important to find a data distribution using a goal-oriented approach. A data distribution is desirable that balances the computational load evenly among the processes while, at the same time, minimizes the communication among the processes.

The problem of finding a data distribution is also interesting from another perspective of undergraduate teaching. It offers the opportunity to demonstrate that a theoretical model can serve as a successful abstraction of a practical problem. More precisely, a formal approach using concepts from graph theory is capable of tackling the data distribution problem systematically.

To this end, we now assume that the nonzero pattern of the asymmetric matrix A is symmetric. Then, the matrix can be represented by an undirected graph $G = (V, E)$. The set of nodes $V = \{1, 2, \ldots, N\}$ is used to associate a node to every row (or corresponding column) of A. The set of edges

$$E = \{(i, j) \mid i, j \in V \text{ and } a_{ij} \neq 0 \text{ for } i > j\}$$

describes the nonzero entries. Here, the condition $i > j$ indicates that the edge (i, j) is identical to the edge (j, i) and that there is no self-loop in G. The data distribution to p processes is then represented by the partition

$$P : V \to \{1, 2, \ldots, p\}$$

that decomposes the set of nodes V of the graph into p subsets V_1, V_2, ..., V_p such that

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

with $V_i \cap V_j = \emptyset$ for $i \neq j$.

Then, (4.4) is reformulated in terms of graph terminology by

$$y_i = a_{ii} \cdot x_i + \sum_{\substack{(i,j) \in E \\ P(i) = P(j)}} a_{ij} \cdot x_j + \sum_{\substack{(i,j) \in E \\ P(i) \neq P(j)}} a_{ij} \cdot x_j.$$

Here, the first two terms of the right-hand side can be computed on process $P(i)$ without communication to any other process. The condition $P(i) \neq P(j)$ in the last term shows that the computation of $a_{ij} \cdot x_j$ requires communication between process $P(i)$ which stores a_{ij} and process $P(j)$ which stores x_j. Minimizing interprocess communication then roughly corresponds to minimizing the number of edges connecting nodes in different subsets V_i of the partition P. This number of edges is called the *cut size* and is formally defined by

$$\text{cutsize}(P) = \big|\{(i, j) \in E \mid P(i) \neq P(j)\}\big|. \tag{4.6}$$

In this graph model, the cut size does not exactly correspond to the number of words communicated between all processes in the computation of $\mathbf{y} \leftarrow A\mathbf{x}$

for a given partition P. However, it gives a reasonable approximation to this amount of communicated data called the *communication volume*; see the corresponding discussion in [59]. The communication volume is exactly described by the cut size if the underlying model is changed from an undirected graph to a hypergraph [60, 61, 62].

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 ε-balanced defined as

$$\max_{1 \leq i \leq p} |V_i| \leq (1 + \varepsilon) \frac{|V|}{p}, \tag{4.7}$$

for some given $\varepsilon > 0$. The graph partitioning problem consists of minimizing the cut size of an ε-balanced partition. It is a hard combinatorial problem [63].

To illustrate the connection between computing a sparse matrix-vector multiplication in parallel and partitioning an undirected graph, we propose a novel educational module. This module is part of a growing set of educational modules called EXPLoring Algorithms INteractively (EXPLAIN). This collection of web-based modules is designed to assist in the effectiveness of teachers in the classroom and we plan to make it publicly available in the near future. Figure 4.17 shows the overall layout of this interactive module for sparse matrix-vector multiplication. The top of this figure visualizes—side by side—the representation of the problem in terms of the graph G as well as in terms of the matrix A and the vector $\mathbf{x}$. 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 score diagram recording values characterizing communication and load balancing. At the bottom part, there are input controls used to select a matrix from a predefined set of matrices, to upload a small matrix, and to choose the layout of the graph vertices.

The first 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 4.18 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

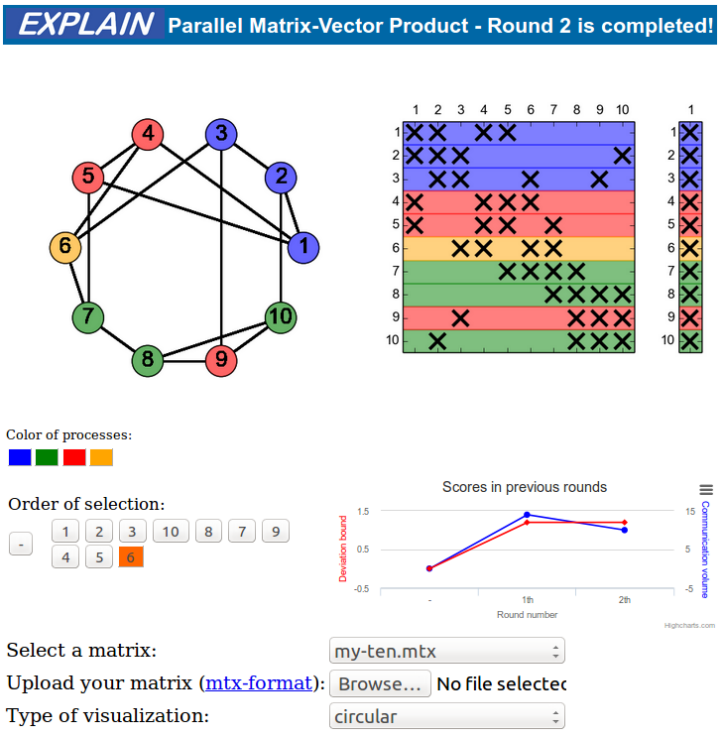


Figure 4.17: Overall structure of the sparse matrix-vector multiplication module.

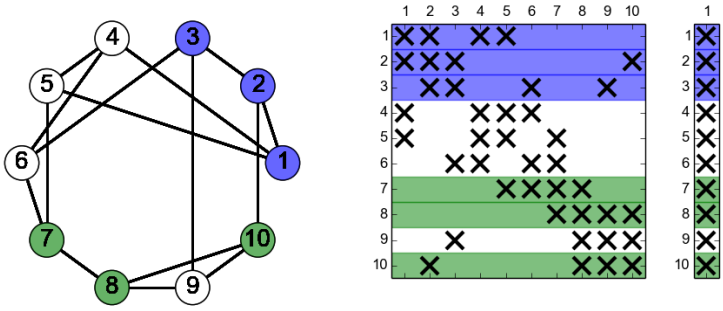


Figure 4.18: The intermediate state after the student selected six vertices.

color j and then clicking on a certain number of vertices, say $i_1, i_2, \ldots, i_s$ such that $P(i_1) = P(i_2) = \cdots = P(i_s) = j$. 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 4.18 illustrates the situation after the student distributed vertices 1, 2 and 3 to the blue process and the vertices 7, 8 and 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 $\mathbf{x}$ are simultaneously colored with the same color. This way, the data distribution is visualized in the graph and in the matrix simultaneously which emphasizes the connection between the matrix representation and the graph representation of that problem. The panel labeled "Order of selection" records the order of the vertices that are interactively chosen. By inspection from that panel in Figure 4.3, we find out that the status depicted in Figure 4.18 is an intermediate step of the interactive session that led to the data distribution in Figure 4.3. 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 form that state.

In EXPLAIN, the term "round" refers to the process of solving a single instance of a given problem. 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 in Figure 4.3. Then, the student can explore the data distribution in more detail by clicking on a color in the panel labeled "Color of processes." Suppose

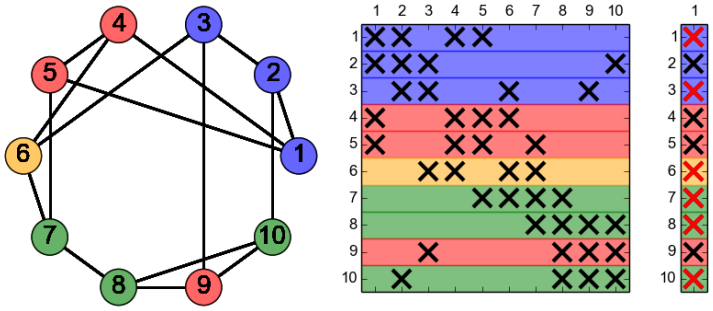


Figure 4.19: All vector entries x_i to be communicated to the red process are drawn in red.

that the student chooses the red process, then this action will modify the appearance of the vector $\mathbf{x}$ in the matrix representation to the state given in Figure 4.19. Here, all vector entries that need to be communicated to the red process are now also colored red. The background color still represents the process that stores that vector entry. This illustrates, for instance, that the vector entry x_1 is communicated from the blue process to the red process when computing $A\mathbf{x}$ using this particular data distribution. The matrix representation visualizes the reason for this communication. There is at least one row that is colored red and that has a nonzero element in column 1. In this example row 4 and row 5 satisfy this condition. Thus, x_1 is needed to compute y_4 and y_5. Again, EXPLAIN visually illustrates the connection between the linear algebra representation and the graph representation. In the graph representation, all vector entries that need to be communicated to the red process correspond to those non-red vertices that are connected to a red vertex. In this example, this condition is satisfied for vertices 1, 3, 6, 7, 8, 10 which corresponds to the vector entries x_1, x_3, x_6, x_7, x_8, x_{10} in the matrix representation that are drawn in red.

When a round is completed 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. To asses these two quantities, the module introduces the score diagram. An example of a score diagram is depicted in Figure 4.20. For each round, this diagram shows the cut size defined by (4.6) using the label "communication volume." As mentioned in the previous section, the cut size in this undirected graph model is not an exact representation of

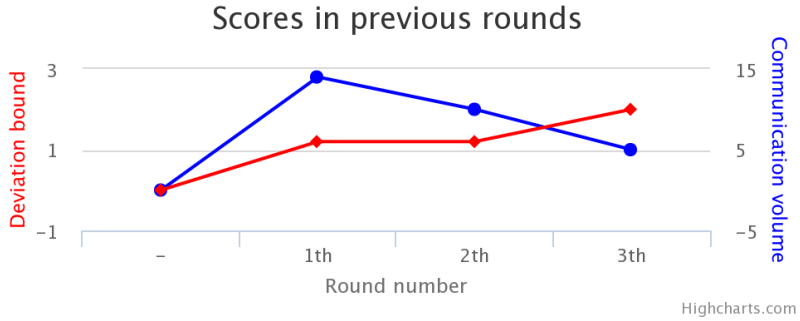


Figure 4.20: The communication volume and the deviation bound versus various rounds.

the communication volume. However, it often captures the nature of the communication volume quite well. Therefore, this graph model uses the cut size as a measure of the communication volume. In that score diagram, the student can check his or her attempt to minimize the communication volume over a number of rounds.

The parameter ε introduced in (4.7) 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 ε the load balancing condition (4.7) is relaxed. The larger ε is chosen, the larger is the allowed imbalance. Thus, in some way, ε quantifies the deviation from a perfect load balance. An equivalent form of (4.7) is given by

$$\frac{p}{|V|} \max_{1 \leq i \leq p} |V_i| - 1 \leq \varepsilon, \tag{4.8}$$

which can be interpreted as follows. Suppose that you are not looking for an ε-balanced partition P for a given ε, but rather turn this procedure around and ask: "Given a partition P, how large need ε at least be so that this partition is ε-balanced?" Then the left-hand side of the inequality (4.8) 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. The score diagram 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 large imbalance of the load. The score diagram helps the student to evaluate the quality of a single

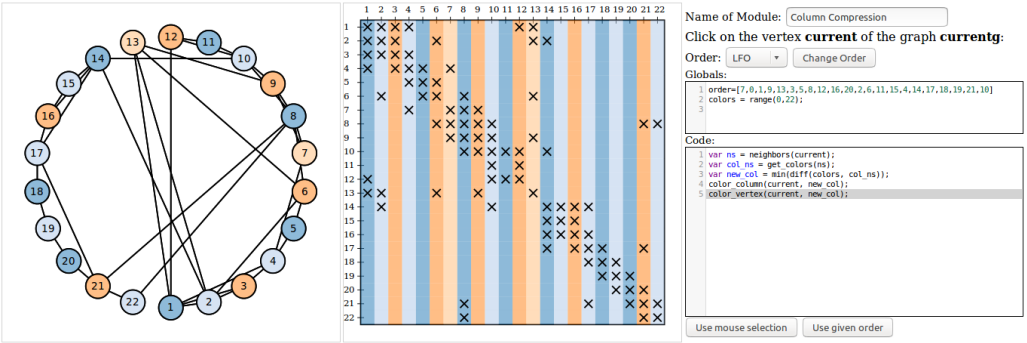


Figure 4.21: The code of the corresponding module is visualized beside graph and matrix. This code is in a simple scripting language. This code can be executed step by step and with the order which is specified by the user.

data distribution and to compare it with distributions obtained in previous rounds. This feedback to the student is designed 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 low deviation bound.

4.6 Preorderings

As we discussed, the ordering is one of the mail elements of heuristics algorithm in both preconditioing and coloring. The process of selection that we have considered in modules of EXPLAIN are actually the illustration of this importance. This is not always doable for students to find a good selection order specially for big matrices.

EXPLAIN 2.0 has a new feature to examine different orders and see the results of algorithm visually. In this feature, different pre-orderings are available which can be selected. A custom order is also possible. Figure 4.21 illustrates graph and matrix and the corresponding source code of the module. This source code is written in a simple scripting language specially designed for EXPLAIN. This source code can be also edited temporarily. Later, the student can decide to start the selection by mouse or an auto-

matic visualization of the algorithm in which the order is defined by the variable "order".

4.7 Implementation Details

4.7.1 EXPLAIN 1.0

The previous version of the software [64] needed a client with administrator privileges to install Python libraries as well as the software itself. In the new implementation, the software is moved to the online platform which means the student needs just a web browser to work with the software. In this section, we shortly describe the underlying algorithm as well as how it is implemented.

The underlying algorithm of coloring and keeping the history is shown in the pseudo-codes given in Figure 4.22. 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. In addition, 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 existing history and new images will be saved.

EXPLAIN combines several Python packages to implement these algorithms. More precisely, the graph data structure is handled by *NetworkX* [65]. It provides different operations like creation and deletion of vertices and edges. It also allows the programmer to access typical graph information such as the neighbor vertices and the number of vertices. Using this library together with *matplotlib* [66] covers the different aspects of visualization. Different layouts of graphs as well as the properties of vertices and edges

```
1: ColorList ← {green, turquoise, orange, violet, ...}
2: Hist ← {}                                   ▷ History of selected vertices.
3: WhereInHist ← 0                             ▷ Where in history are we?
4: IsInHist ← False                            ▷ Are we in history mode?
5:
6: procedure VertexClicked(v)                  ▷ User clicks vertex v.
7:     ns ← neighbors(v)
8:     ColorIndex ← 1                          ▷ Allowed color index
9:     for i ← 1 to size(ColorList) do
10:        AllowedColor ← True
11:        for j ← 1 to size(ns) do
12:            if ColorList[i] = color(ns[j]) then
13:                AllowedColor ← False
14:        if AllowedColor = True then
15:            ColorIndex ← i
16:            Break
17:    Color v with the color ColorList[ColorIndex]
18:
19:    if graph and matrix images are not already saved then
20:        SaveMatrix()                        ▷ Using a specific name
21:        SaveGraph()                         ▷ Using a specific name
22:    if IsInHist = True then
23:        for i ← WhereInHist + 1 to size(Hist) do
24:            Hist.removeElementAtPosition(i)
25:        Hist.add(v)
26:        IsInHist ← False
27:    else
28:        Hist.add(v)
29:    Update(Hist)                            ▷ Update history buttons
30:
31:
32: procedure HistClicked(h)                   ▷ User clicks history h.
33:    OpenMatrix(h)                           ▷ Plot/load matrix with specific name
34:    OpenGraph(h)                            ▷ Plot/load graph with specific name
35:    WhereInHist ← find(Hist, h)             ▷ The location of h
36:    IsInHist ← True
```

Figure 4.22: Pseudocode of the event handling in EXPLAIN

are produced by this library. The matrix manipulation and visualization is handled by *Scipy* [67], 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* [68]. 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. As a result, the Python code can be run as a web application.

Previously, 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 client side shows HTML websites with embedded Javascript source code while the server side is written in Python. Since the buttons are HTML buttons and the events are Javascript functions, a form is submitted to the Python server by a Javascript function 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.

4.7.2 EXPLAIN 2.0

As we discussed, 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* [65] for the graph data structure, *matplotlib* [66] for the visualization aspects, *Scipy* [67] for the sparse matrix computation, and *Mod_python* [68] 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 system.

In the new implementation, we replaced all python parts with the JS code.

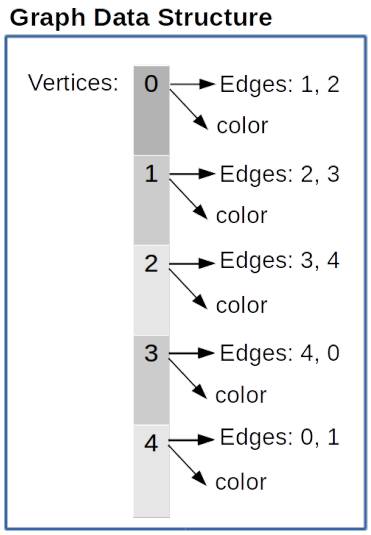


Figure 4.23:

We decided for the Javascript library D3 (Data-Driven Documents) because of its power of control and visualization. The adjacency list is selected for the graph data structure. We used the object structure of the JS which helps us to have another properties also in the graph. Figure 4.23 shown an illustration of the data structure. There is basically an array representing the vertices. Each cell of array points to another array *edges* which contains the id numbers of the vertices which are neighbours of that cell. Here, we showed that the data structure can contain another properties like colors. It can be extended dynamically to contain any other properties which is needed later. For example, two other properties *distance* and *parent* are added in the implementation of Breadth-First Search (BFS).

Another important aspect which is considered in our design was the connection of the model and view. An effective approach enables the direct change in view while keeps the separation of the view and model. So, we decided to have unique ids for the view of edges and vertices. The unique ids for vertices are defined as the concatenation of the string "ver" and the actual id of vertex. Similarly, the unique ids for edges are defined as the concatenation of the four string "edge", the source vertex, "-", and the target vertex. The same idea is used for the matrix view. Each cell of the matrix is accessible through an unique id combining the strings "cell", the row index,

neighbors(l_v)	returns the neighbours of the vertex l_v
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 difference of two given sets A and B
get_colors(l_v)	return a list of colors of vertices l_v

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

"-", and the column index. Each nonzero at matrix also gets the unique id same as before while inserting the string "nnz" instead of cell.

The previous discussion of the view access enables us to select the specific element and change its behaviour and properties. For example, the following code would change the color of the vertex with the id i and the color of a matrix cell,

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

There are two important aspects of implementation which we discuss here. An aspect of implementation is the real connection of an edge to its source and target vertices. It means that the locations of the source and target vertices are read directly from the vertex component in view. This prevents of the double repaint the view. Another aspect is to paint the components in the correct order. So, no edges are viewed about vertices or no cells of matrix are shown above the nonzeros. We did this by using the group component of D3 in which it can be managed to be drawn always in the given order.

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 decided to have a new script language which has only functionality regarding the matrix and the graph and their connections. Also, this new script language does everything by simple functions such that each function does a specific action on the graph or matrix. Table 4.1 shows a list of functions which are available now in EXPLAIN 2.0.

Having such scripting language empowers us to have a dynamic script editor together with EXPLAIN which makes the creation of new module very 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* represents the current vertex. The following code shows the code for the column compression module as an instance. The code is very shorter than the implementation in EXPLAIN 1.0.

```
1 var ns = neighbors(current);
2 var col_ns = get_colors(ns);
3 var new_col = min(diff(colors,col_ns));
4 color_column(current, new_col);
5 color_vertex(current,new_col);
```

Chapter 5

Conclusion and Future Work

This work is to develop the new ideas in combinatorial scientific computing. Our focus is the new idea of mixing AD and preconditioning which we explained in Section 2.3. The whole idea is to save storage as well as to optimize the convergence of iterative linear solvers. We introduce new heuristics here for a better convergence and to increase the number of elements of the matrix which appear in preconditioner while the memory usage for the Jacobian computation remains almost the same. Additionally, we extend an interactive educational module toward teaching the concepts of combinatorial scientific computing. We added a lot of new features and concepts to this module.

Although there is much existing research, many dimensions of the problem can be improved. Most of the works toward mixing AD and preconditioning looks at solutions for a general matrix. However, the study of Hessians' matrix and its connection to fill-in of Cholesky factorization might offer more degree of freedom in coloring.

On the other hand, we considered only the ILU preconditioning. What if we replace ILU with the approximate inverse preconditioning (AINV) [69] which produces no fill-in. There is a need to review the AINV algorithms. An algorithm for this technique takes a pattern for preconditioner M as input and gradually updates this starting pattern and the nonzero values. So we can do first a sparsification as usual which leads to R_p and take $R_i + R_p$ as starting pattern for M. Then, we would have two separate problems: coloring and AINV preconditioning. It means that two matrices R_init and M are provided as input. We then color the graph with this information. It is a different coloring problem than the usual partial coloring problem. However,

this seems to be the same as partial coloring with a larger set of required elements. When applying AINV to an initial M, is there any effect on M? For instance, does the pattern of M vary? How does it vary? Does the variation depend on the structure of the initial pattern only? Or does the pattern vary based on the values of M that are available at runtime? Perhaps, one can estimate/approximate the pattern of M conservatively without taking the values of M into account. (This would then be similar to the fill-in when using ILU.) Need to look at AINV algorithms more closely. What if we transform program for function $f(x)$ not only to a program for $\frac{df}{dx}$ (traditional AD), but also to another program for matrix-vector product Mx where M is an AINV preconditioner for the Jacobian $\frac{df}{dx}$.

Another future work is to consider blocks of submatrices for coloring instead of just a complete row or column? Steihaug and Hossain [70] discuss this idea for blocks of rows and the same column intersection graph model as before and show it has advantages in the unidirectional coloring. A first improvement is to search for the equivalent approach in the bidirectional coloring and partial 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. 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.

[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] 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.

[6] ——, "The efficient computation of sparse Jacobian matrices using automatic differentiation," *SIAM Journal on Scientific Computing*, vol. 19, no. 4, pp. 1210–1233, 1998.

[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] 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.

[9] 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/

[10] 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

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

[12] Y. Saad, *Iterative Methods for Sparse Linear Systems*. Boston: PWS Publishing Company, 1996.

[13] 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," *Conference on Combinatorial Scientifi Computing*, 2016.

[14] 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.

[15] 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.

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

[17] E. G. Ng and P. Raghavan, "Performance of greedy ordering heuristics for sparse cholesky factorization," *SIAM J. Matrix Anal. Appl.*, vol. 20, no. 4, pp. 902–914, Jul. 1999. [Online]. Available: http://dx.doi.org/10.1137/S0895479897319313

[18] S. Ma and Y. Saad, "Distributed ilu(0) and sor preconditioners for unstructured sparse linear systems," SIAM Journal on Computing, Tech. Rep., 1998.

[19] P. R. Amestoy, T. A. Davis, and I. S. Duff, "An approximate minimum degree ordering algorithm," *SIAM J. Matrix Anal. Appl.*, vol. 17, no. 4, pp. 886–905, Oct. 1996. [Online]. Available: http://dx.doi.org/10.1137/S0895479894278952

[20] A. George and J. W. Liu, *Computer Solution of Large Sparse Positive Definite*. Prentice Hall Professional Technical Reference, 1981.

[21] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM J. Sci. Comput.*, vol. 20, no. 1, pp. 359–392, Dec. 1998.

[22] D. LaSalle and G. Karypis, *Efficient Nested Dissection for Multicore Architectures*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 467–478.

[23] J. P. Spinrad and G. Vijayan, "Worst case analysis of a graph coloring algorithm," *Discrete Applied Mathematics*, vol. 12, no. 1, pp. 89 – 92, 1985. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0166218X85900435

[24] 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

[25] 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

[26] 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

[27] J. R. Allwright, R. Bordawekar, P. D. Coddington, K. Dincer, and C. L. Martin, "A comparison of parallel graph coloring algorithms," 1995.

[28] 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.

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

[30] D. Brélaz, "New methods to color the vertices of a graph," *Commun. ACM*, vol. 22, no. 4, pp. 251–256, Apr. 1979.

[31] M. Lülfesmann, "Graphfärbung zur partiellen Berechnung von Jacobi-Matrizen," Master's thesis, Department of Computer Science, RWTH Aachen University, Aachen, Germany, 2006.

[32] A. K. M. S. Hossain and T. Steihaug, "Computing a sparse jacobian matrix by rows and columns," 1995.

[33] A. Calotoiu, "Bipartite Graph Coloring for Compressed Sparse Jacobian Computation," Master's thesis, University Politehnica of Bucharest, Bucharest, Romania, 2009.

[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] J. J. Camata, A. L. G. A. Coutinho, A. M. P. Valli, L. Catabriga, and G. F. Carey, "Block ilu preconditioners for parallel amr/c simulations."

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

[38] 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

[39] 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

[40] R. Lischner, *Documentation*. Berkeley, CA: Apress, 2013, pp. 173–179. [Online]. Available: http://dx.doi.org/10.1007/978-1-4302-6194-0_27

[41] 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.

[42] 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.

[43] M. A. Rostami, H. M. Bücker, and A. Azadi, "A new approach to visualizing general trees using thickness-adjustable quadratic curves," in *Graph Drawing, Proceedings of 22nd International Symposium on Graph Drawing, GD 2014, Würzburg, Germany, September 24–26, 2014*, ser. Lecture Notes in Computer Science, C. Duncan and A. Symvonis, Eds., vol. 8871. Berlin, Germany: Springer, 2014, pp. 525–526, extended abstract for poster.

[44] T. Mansour, M. A. Rostami, E. Suresh, and G. B. A. Xavier, "New sharp lower bounds for the first Zagreb index," *Scientific Publications of the State University of Novi Pazar*, vol. 8, no. 1, pp. 11–19, 2016.

[45] T. Mansour, M. A. Rostami, S. Elumalai, and B. A. Xavier, "Correcting a paper on Randić and geometric-arithmetic index," *Turkish Journal of Mathematics*, 2016.

[46] T. Mansour, M. A. Rostami, E. Suresh, and G. B. A. Xavier, "On the bounds of the first reformulated Zagreb index," *Turkish Journal of Analysis and Number Theory*, vol. 4, no. 1, pp. 8–15, 2016. [Online]. Available: http://pubs.sciepub.com/tjant/4/1/2

[47] ——, "A short note on hyper Zagreb index," *Boletim da Sociedade Paranaense de Matemática*, 2016.

[48] 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.

[49] 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.

[50] 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.

[51] ——, "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.

[52] 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

[53] 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.

[54] 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

[55] 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

[56] 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

[57] 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.

[58] 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.

[59] B. Hendrickson and T. G. Kolda, "Graph partitioning models for parallel computing," *Parallel Computing*, vol. 26, no. 2, pp. 1519–1534, 2000.

[60] Ü. V. Çatalyürek and C. Aykanat, "Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication," *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, no. 7, pp. 673–693, 1999.

[61] Ü. V. Çatalyürek, B. Uçar, and C. Aykanat, "Hypergraph partitioning," in *Encyclopedia of Parallel Computing*, D. Padua, Ed. Springer US, 2011, pp. 871–881.

[62] B. Uçar and C. Aykanat, "Revisiting hypergraph models for sparse matrix partitioning," *SIAM Review*, vol. 49, no. 4, pp. 595–603, 2007. [Online]. Available: http://dx.doi.org/10.1137/060662459

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

[64] 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.

[65] 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.

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

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

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

[69] 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.

[70] 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.