

University Politehnica of Bucharest  
Faculty of Automatic Control and Computers  
Computer Science and Engineering Department

Diploma Thesis

# **Bipartite Graph Coloring for Compressed Sparse Jacobian Computation**

by

**Alexandru Calotoiu**

Supervisor: S.I. Dr. Ing. Emil Slușanschi

Supervisor: PD Dr. Ing. Martin Bucker

Assistant Supervisor: Dipl. Inf. Michael Lulfesmann

Bucharest, June/July 2009



---

# Contents

---

<b>Contents</b>	<b>i</b>
<b>1 Introduction</b>	<b>3</b>
<b>2 Previous work and state of the art</b>	<b>7</b>
2.1 Derivative computation . . . . .	7
2.1.1 Symbolic derivation . . . . .	8
2.1.2 Finite differences . . . . .	8
2.1.3 Automatic differentiation . . . . .	9
2.2 Seeding techniques . . . . .	13
2.3 Graph coloring . . . . .	17
2.3.1 Preliminary concepts and notions . . . . .	17
2.3.2 Graph coloring . . . . .	18
2.3.3 Bipartite graphs . . . . .	20
2.3.4 Graph coloring framework . . . . .	23
2.3.5 Testing . . . . .	23
2.3.6 Challenges . . . . .	25
2.4 Algorithm for graph coloring - overview of issues . . . . .	25
2.5 Partial Jacobian computation . . . . .	27
2.5.1 Motivation . . . . .	27
2.5.2 Concepts . . . . .	28
<b>3 Hardware and Software Configuration</b>	<b>31</b>
3.1 Hardware . . . . .	31
3.2 Software . . . . .	33
3.3 Project implementation . . . . .	33
3.3.1 Preliminary issues . . . . .	33

<b>4 Two-sided coloring algorithms for complete Jacobian matrix computation</b>	<b>35</b>
4.1 Star bicoloring scheme . . . . .	36
4.2 Complete direct cover . . . . .	38
4.3 Row/Column Fill Bicoloring Algorithm . . . . .	41
4.4 Star bicoloring algorithm with restriction relaxation and queue system . . . . .	43
4.5 Refinement of results using postprocessing methods . . . . .	45
4.6 Integrated star bicoloring scheme . . . . .	47
4.7 Total ordering star bicoloring algorithm . . . . .	51
<b>5 Restricted coloring for partial Jacobian computation</b>	<b>55</b>
5.1 Restricted star bicoloring algorithm . . . . .	55
5.2 Restricted Total Ordering Star Bicoloring Algorithm . . . . .	57
5.3 The main diagonal pattern . . . . .	58
5.4 The lattice pattern . . . . .	58
5.5 The high-value element pattern . . . . .	59
5.6 The diagonal block . . . . .	59
5.7 Results . . . . .	60
<b>6 Conclusions</b>	<b>65</b>
<b>7 Future work</b>	<b>69</b>
<b>Bibliography</b>	<b>71</b>
<b>List of Figures</b>	<b>73</b>
<b>List of Tables</b>	<b>75</b>
<b>Listings</b>	<b>76</b>

---

# Abstract

---

*Abstract* Graph coloring as a method for computing sparse Jacobian matrices more efficiently has been successfully employed since the 1980s. This thesis begins with an introduction into the field of scientific computing, automated differentiation and graph coloring followed by a description of the state of the algorithms in this field. The goal of the thesis is to present the improvements made to the existing graph coloring algorithms and the research done in restricted coloring for partial Jacobian computation. A description of the algorithms developed to improve computation along with extensive test results is offered. Patterns used in restricted bidirectional Jacobian computation are presented.

*Key words* Jacobian matrix. Graph Coloring. Automatic Differentiation. Scientific Computing. Sparse matrix. Restricted Jacobian Computation.



# Chapter 1

---

## Introduction

---

Scientific computing is a large research field with many complex and interesting problems. Some of these problems, though researched to different degrees are still open even, such as the domain of graph coloring for sparse Jacobian computation. Many scientific problems involve solving a system of linear equations of the form

$$Az = b \tag{1.1}$$

where  $A \in \mathbb{R}^{n \times n}$  is a large, sparse matrix, and  $z, b \in \mathbb{R}^n$ . Such systems are needed for solving partial differential equations along with finite difference methods. The matrix  $A$  can for example represent a Jacobi matrix  $J|_{x_0}$  in a point  $x_0 \in \mathbb{R}^n$ . These can be used for example in computational fluid dynamics, CFD.

Systems of linear equations can be solved with direct or iterative methods. While giving just an approximate solution an iterative method is often preferred as it is faster to compute and needs less memory space. Direct methods offer exact solutions but at the cost of computational time and increased storage requirements.

The use of preconditioners can speed up the computing of solutions of linear equation systems even more. The Matrix  $M$ , is created for this purpose. As an example, using left-hand side preconditioning applied to the above system yields this equation system:

$$M^{-1}Az = M^{-1}b. \tag{1.2}$$

The preconditioner  $M$  can be determined out of a subset of the elements of  $A$ . Thus it is not always required to discover all elements of  $A$ . An example for this is the Newton method.

The function  $f$ , whose first derivative is  $J$ , can be supplied in the form of a computer program. Ways to compute this matrix can be symbolic derivation, finite differences and *automatic differentiation*. Automatic differentiation transforms the original function by adding the directional derivative, a column of the Jacobi matrix, or the adjoint, a row of the same in addition to computing the result of the functions itself.

With *seeding techniques* it is possible to compute more columns or rows using just one iteration of AD (Automatic Differentiation), thus reducing the number of AD runs required to compute the entire Jacobian and as such fewer resources needed. Given the sparsity structure the seed matrix must be determined, to find out which rows and/or columns can be computed at once while still allowing for a direct determination of the Jacobian elements from the result. In the case of sparse matrices this brings great improvements in performance.

The idea is that if the matrix is sparse then it is for example possible to compute two columns of the matrix at once if there are no nonzero elements on the same row. A set of columns compressed this way is called a partition. This is also possible for rows. It is possible to create partitions for both rows and columns and as will be shown in this work, have a smaller number of partitions on the whole.

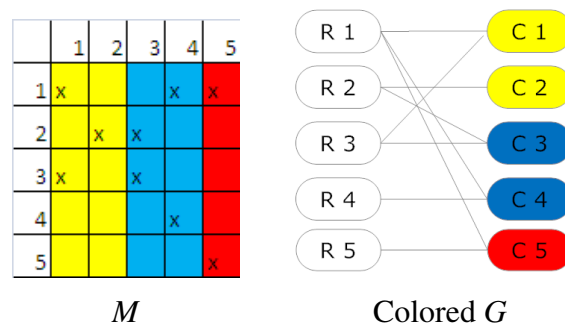


Figure 1.1: Unidirectional coloring of a matrix and bipartite graph. For this coloring, complete unidirectional column coloring, the condition that must be upheld is that no two nodes that share a neighbor can have the same color.



Modeling the Jacobian as a bipartite graph, and then coloring the resulting graph as shown in Figure 1.1 is a way of creating the partitions represented by the combined sets of rows and columns. The heuristics involved are of the major issues on which this work focuses. The coloring of the bipartite graph can be unidirectional or bidirectional, meaning that either rows, columns or both rows and columns are compressed. For example, by coloring the rows 1 and 2 with a color  $a$ , the columns 1, 4 and 5 with a color  $b$ , the column 3 with a color  $c$  a valid coloring scheme is obtained.

The coloring discussed so far results in a determination of the entire Jacobian matrix. There are cases where this is not necessary. Patterns can be available or can be determined to mark those nonzero elements that are required and a coloring scheme can be devised to obtain just those values. Finding new patterns and new heuristics in this area is an interesting challenge, the work already done in this field leaving many subjects still open for study. The partial computation of the Jacobian can be done by using restricted coloring of the bipartite graph, either unidirectional or bidirectional.

This work focuses on adapting and improving the state of the art heuristics for full Jacobian computation and partial Jacobian computation. This is done by adapting different already existing graph coloring theories to this particular problem and combining the advantages of these methods with the already implemented state of the art algorithm. Tests have also been performed using new patterns that can be used in restricted bidirectional coloring and assessing the results and improvements of the heuristics implemented using these patterns.



## Chapter 2

---

# Previous work and state of the art

---

### 2.1 Derivative computation

The solving of many scientific simulation problems involves the computation of the derivatives. Thus, accurate and fast computation of derivatives has been an ongoing field of study for more than 30 years. A use for derivatives is the Newton method for solving a non-linear function. The goal of many scientific simulations is optimization of various processes, which contributes to improving performance.

A well-known technique is to use derivatives to find a maximum and a minimum of a function hence the need for the computation of first order derivatives. Given the fact that many functions used in scientific simulation are quite complex, derivative computation becomes very time consuming. Given a function

$$f(x) = y, \quad f : \mathbf{R}^n \rightarrow \mathbf{R}^m, \quad x \in \mathbf{R}^n, \quad y \in \mathbf{R}^m \quad (2.1)$$

a great number of derivatives need to be computed even to determine the derivative of  $f$  in a single point. Computing the Jacobian matrix for the function  $f$  in  $x_0$ , means calculating

$$f'_{i,j}|_{x=x_0} = \left. \frac{\partial y_j}{\partial x_i} \right|_{x=x_0} \quad (2.2)$$

for every  $i, j$  which results in  $n \times m$  derivatives for each point.

In the computation of derivatives three methods have emerged: symbolic differentiation, finite differences and automatic differentiation.

### 2.1.1 Symbolic derivation

*Definition* Symbolic differentiation [Spi83] involves determining the analytical derivative of a function, and the value of the derivative at any point can be subsequently calculated by simply substituting the value into the formula obtained. Although this method is the closest to the way humans determine derivatives ‘by hand’, this method is not very suited for computer-aided usage.

*Disadvantages* There exist computer algebra systems such as Maple which are able to work with symbolic differentiation but the limitations are too great to be widely used. Any simulation program that would require symbolic derivation needs to be transformed into explicit formula form, which for imperative programming elements such as loops or decisions is time consuming and complicated.

### 2.1.2 Finite differences

*Definition* Finite differencing [Boo07] is one of the methods employed to numerically solve differential equations. It plays a central role in this area due to it being an established method with a longer history than automatic differentiation. This will hopefully change with the increasing attention that automatic differentiation receives. As a method to determine the derivative, the finite difference is the approximation of the limes represented by the derivative in a fixed point. As such, it is a mathematical expression of the form  $f(x+b) - f(x+a)$ . Divided by  $b - a$ , a difference quotient is obtained. Considering that the derivative of a function  $f$  at a point  $x$  is defined by the limit

$$f'(x) = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h}. \quad (2.3)$$

one can use  $h$  with a fixed (nonzero) value, instead of approaching zero, to obtain

$$\frac{f(x+h) - f(x)}{h}. \quad (2.4)$$

This difference approximates the derivative when  $h$  is small. The error in this approximation can be determined using Taylor’s theorem, and is

$$\frac{f(x+h) - f(x)}{h} - f'(x) = O(h) \quad (h \rightarrow 0). \quad (2.5)$$

*Disadvantages* The computation is quite easy, as determining the values of the function in two points suffice. Determining the value of  $h$  so that the error is minimal, which usually implies a small  $h$ , but also to avoid any underflow errors which means  $h$  must be big enough to avoid errors caused by the limited precision of floating point arithmetic computers, is a much bigger issue and a lot of work has been done in determining methods of choosing a ‘good’  $h$ . Another disadvantage is that the finite difference methods does not allow the reverse mode.

### 2.1.3 Automatic differentiation

*Definition* Automatic differentiation (AD)[AG08, Ra181] also called algorithmic differentiation, is a method to numerically evaluate the derivative of a function specified by a computer program. AD exploits the fact that any computer program that implements a function  $y = f(x)$  can be decomposed into a sequence of elementary assignments

An elementary assignment is any statement of the form  $x = op \ a$  or  $x = a \ op \ b$  where  $op$  can only be a simple operator such as  $+$ ,  $*$ , etc or the form  $x = f(a)$  where  $f$  is an elementary function such as  $\sin, \cos$ , any one of which may be trivially differentiated by a simple table lookup.

These are elemental partial derivatives. Partial derivatives are derivatives of a function with multiple inputs where one of the parameters is varied whilst the others are constant. Evaluated at a particular argument and combined in accordance with the chain rule from derivative calculus to form some derivative information for  $f$  (such as gradients, the Jacobian matrix, etc.). This process yields exact (to numerical accuracy) derivatives. Because the symbolic transformation occurs only at the most basic level, AD avoids the computational problems inherent in complex symbolic computation.

*Overview* AD can be described as a black box, with inputs and outputs. The inputs of AD are a function and a direction for the derivation. Its output will consequently be a modified function, which has the same results as the original with the additional output of the derivatives in the required direction.

AD is based on the decomposition of differentials provided by the chain rule. Given the function  $f(x) = g(h(x))$  AD uses the fact that

$$\frac{df}{dx} = \frac{dg}{dh} \frac{dh}{dx}. \quad (2.6)$$

Two modes of AD exist, forward mode and reverse mode. In the example above, forward mode means computing first  $dh/dx$  and then  $dg/dh$ , while reverse mode does the opposite. Both methods will be explained into detail later. From now on we shall consider  $J$  the Jacobian  $J|_{x=x_0}$  for the original function 2.1.

*Jacobian Computation* The Jacobian  $J$  of  $y = f(x) : \mathbf{R}^n \rightarrow \mathbf{R}^m$  is an  $m \times n$  matrix, where the element at  $J_{i,j} = \frac{\partial y_i}{\partial x_j}$ . The Jacobian can be computed using  $n$  sweeps of forward mode, of which each sweep can yield a column vector of the Jacobian, or with  $m$  sweeps of reverse mode, of which each sweep can yield a row vector of the Jacobian.

### Forward mode

The forward mode means evaluating derivatives of functions in an input to output use of the chain rule and is particularly efficient if there are significantly fewer input variables than output variables. An advantage of the forward mode is that it can be realized at the same time as the normal run of the program.

There are two methods of using forward mode, operator overloading (OO) and source transformation (ST). Operator overloading, as its names suggests involves modifying all operators so that they compute the derivatives in addition to their normal actions.

ST involves adding a transformation of the original function which calculates a directional derivative, where  $\dot{x} \in \mathbf{R}^n$  is initialized with each eigenvector  $e_i = (0, \dots, 1, \dots, 0)^T$ ,  $i = 1, \dots, n$  to determine  $J$ . For each vector  $\dot{x} = e_i$  the derivative  $\dot{y}$  is computed. The vector  $\dot{x} = e_i$  is called a seed vector. The vector  $\dot{y}$  represents the  $i$ -th column of the Jacobian.

$$\dot{f}(x, \dot{x}) = f'(x) * \dot{x} = \dot{y}, \quad \dot{f} : \mathbf{R}^{2n} \rightarrow \mathbf{R}^m \quad (2.7)$$

*Example* Let us look at an example of how a function is computed using forward mode of AD. Let the function be

$$f(x_1, x_2) = x_1 * x_2 + \cos(x_1) \quad (2.8)$$

and let the point  $x = (x_1, x_2)$  where we will compute the derivative be  $(1; 0, 5)$ .

Let the variables  $v_{1-n}, \dots, v_0$  be the input variables  $x_1, \dots, x_n$  of the function and  $v_1, \dots, v_m$  be the intermediate and output variables  $y_1, \dots, y_m$  of the function.

The goal is to obtain the derivative objects  $\dot{v}_j$ ,  $j \in 1, \dots, m$ .

First, I will explain the original function evaluation and how it is translated into a computational graph given in the Figure 2.1.

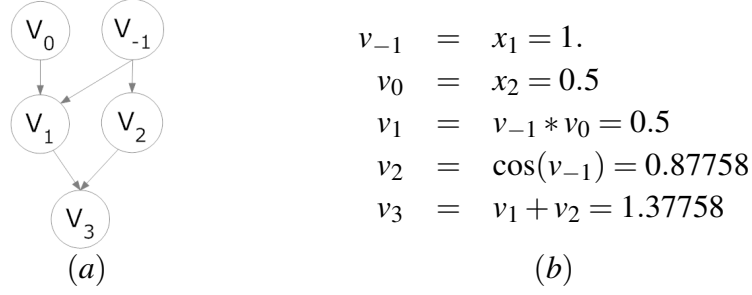


Figure 2.1: (a) Computational graph for  $f(x_1, x_2) = x_1 * x_2 + \cos(x_1)$ ; (b) Computation steps for  $f(x_1, x_2) = x_1 * x_2 + \cos(x_1)$

*Applying forward AD* To determine the derivation objects, first one must compute the partial derivative  $c_{i,j} = \frac{\partial \varphi_j}{\partial v_i}(v_k)_{k \prec j}$  for each elementary assignment  $\varphi_j(v_i)_{i \prec j}$ . This will create a column-wise computation of the Jacobian. The operator  $\prec$  represents precedence. Applying the Table 2.1 to the computation graph we can

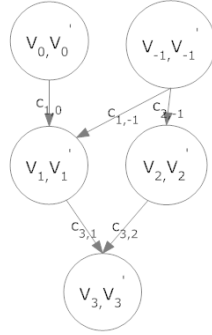
$\varphi_j$	$c_{j,i_1} = \frac{\partial \varphi_j}{\partial v_{i_1}}$	$c_{j,i_2} = \frac{\partial \varphi_j}{\partial v_{i_2}}$
+	1	1
*	$v_{i_2}$	$v_{i_1}$
cos()	$-\sin(v_{i_1})$	—

Table 2.1: Partial derivative operations  $+$ ,  $*$  and  $\cos()$

determine the derivative object  $\dot{v}_i$ . We simply compute it at the same time as  $v_i$ , as can be observed in Figure 2.2. In the example the derivative for the direction  $(1;0)$  will be computed. Every time a vertex in the computational graph is computed, the derivative for that vertex will also be computed using the elementary assignment lookup table and the values previously determined. In the example  $\dot{v}_1$  is determined using the elementary assignment lookup table for the values of  $c_{1,0}$  and  $c_{1,-1}$  and the values previously computed for  $\dot{v}_{-1}$  and  $\dot{v}_0$ . It is obvious now that the forward mode can be easily employed at the same time as the run of the original code. It is quite intuitive and can easily be implemented. The forward mode has seen many implementations such as the Taped mode [Tap], OpenAD [Ope] and ADIFOR [Adi].

### Reverse mode

The reverse mode has advantages over the forward mode if there are a lot fewer output variables when compared to input variables. The reverse mode will be summarily presented here as the complete presentation is beyond the scope of



(a)

$$\begin{aligned}
 v_{-1} &= x_1 = 1. \\
 \dot{v}_{-1} &= \dot{x}_1 = 1. \\
 v_0 &= x_2 = 0.5 \\
 \dot{v}_0 &= \dot{x}_2 = 0. \\
 v_1 &= v_{-1} * v_0 = 0.5 \\
 \dot{v}_1 &= c_{1,0} * \dot{v}_{-1} + c_{1,-1} * \dot{v}_0 \\
 &= 0.5 * 1 + 1 * 0 = 0.5 \\
 v_2 &= \cos(v_{-1}) = 0.87758 \\
 \dot{v}_2 &= c_{2,-1} * \dot{v}_{-1} = -0.47982 * 1 = -0.47982 \\
 v_3 &= v_1 + v_2 = 1.37758 \\
 \dot{v}_3 &= c_{3,1} * \dot{v}_1 + c_{3,2} * \dot{v}_2 \\
 &= 1 * 0.5 + 1 * -0.47982 = 0.02018
 \end{aligned}$$

(b)

Figure 2.2: (a) Computation graph for  $\dot{f}(x_1, x_2) = \dot{x}_1 * x_2 + x_1 * \dot{x}_2 - \sin(x_1)$ ; (b) Computation steps for  $\dot{f}(x_1, x_2) = \dot{x}_1 * x_2 + x_1 * \dot{x}_2 - \sin(x_1)$

this work. The reverse mode adds a transformation of the original function which calculates an adjoint,  $\bar{x} \in \mathbf{R}^n$ , where  $\bar{y} \in \mathbf{R}^m$ , the adjoint direction, is initialized with each eigenvector  $e_i = (0, \dots, 1, \dots, 0)^T$ ,  $i = 1, \dots, m$ , to determine  $J$ . Operations in RM are done by reversing the flow of the computational graph and allows the computation of a row of  $J$ .

$$\bar{f}(x, \bar{y}) = (f'(x))^T * \bar{y} = \bar{x}, \quad \bar{f}: \mathbf{R}^{n+m} \rightarrow \mathbf{R}^n \quad (2.9)$$

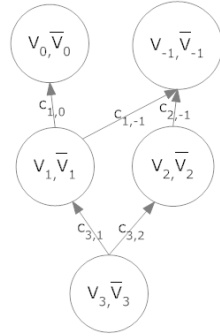
**Example** Let us look at an example of how a function is computed using the reverse mode of AD. Let the function be

$$f(x_1, x_2) = x_1 * x_2 + \cos(x_1) \quad (2.10)$$

and let the point where we wish to compute the derivation be  $(1; 0, 5)$ . Let  $v_{1-n}, \dots, v_0$  be the input variables  $x_1, \dots, x_n$  of the function. Let  $v_1, \dots, v_m$  be the output variables  $y_1, \dots, y_m$  of the function. The goal is to obtain the derivation objects  $\dot{v}_j$ ,  $j = 1, \dots, m$ . The normal computation steps are explained in Figure 2.1

**Applying reverse AD** To compute the adjoint one follows the same steps as in forward mode up to computing partial derivatives and assigning initial values to  $x$ . What follows is the assigning values to  $\bar{y}$  and computing the values of the adjoint objects in reverse order. Using the Table 2.1 and the computational graph we can determine  $\bar{x}_i$  in Figure 2.3. The example will compute the derivative for the direction  $\bar{y} = |1|$





(a)

$$\begin{aligned}
 v_{-1} &= x_1 = 1. \\
 v_0 &= x_2 = 0.5 \\
 v_1 &= v_{-1} * v_0 = 0.5 \\
 v_2 &= \cos(v_{-1}) = 0.87758 \\
 v_3 &= v_1 + v_2 = 1.37758 \\
 \bar{v}_3 &= \bar{y} = 1 \\
 \bar{v}_2 &= c_{3,2} * \bar{v}_3 \\
 &= 1 * 1 = 1 \\
 \bar{v}_1 &= c_{3,1} * \bar{v}_3 \\
 &= 1 * 1 = 1 \\
 \bar{v}_0 &= c_{1,0} * \bar{v}_1 \\
 &= 0.5 * 1 = 0.5 \\
 \bar{v}_{-1} &= c_{1,-1} * \bar{v}_1 + c_{2,-1} * \bar{v}_2 \\
 &= 0.5 * 1 + -0.47982 * 1 = 0.02018
 \end{aligned}$$

(b)

Figure 2.3: (a) Computation graph for  $f(x_1, x_2) = x_1 * x_2 + \cos(x_1)$  with adjoint objects ; (b) Computation steps for  $f(x_1, x_2) = x_1 * x_2 + \cos(x_1)$  with adjoint objects

By looking at the final values in the Figures 2.2 and 2.3 it can be seen that the forward mode and reverse modes get the same results, but the very different styles of determining the values of the Jacobian each method has advantages and disadvantages depending on the nature of the problem solved namely problems with a high input-to-output variable ratio are advantageous for the reverse mode while those with few inputs and many outputs are better suited for the reverse mode.

Many large problems of scientific computing involve the computation of sparse Jacobian matrices. A sparse Jacobian matrix has a significant number of zeros. It is possible to use techniques to compute the Jacobian matrices in fewer passes than normally required. By passes are implied the number of runs of the program that are needed to compute the entire Jacobian. These techniques are called seeding techniques.

## 2.2 Seeding techniques

In practice many problems involve sparse, large  $n \times m$  Jacobian matrices. The sparsity pattern of the Jacobian matrix can be determined by a number of methods such as Bayesian probing [GMM00] Normally, for the computation of a

Jacobian matrix  $n$  passes have to be performed, each time obtaining one column, which translates into a seed matrix  $I_n$ .

The research done by Curtis, Powell and Reed Seeding [AG08] technique allows for compression of these vectors. The idea is to compute more columns in each pass, and thus utilize the sparsity of the matrix. The basis of this method is that two columns can be combined if both have no nonzero element in the same row. If this could be achieved a lot of computing time could be spared by determining more than one directional derivative in one pass. The same can be applied to rows in the case of reverse mode, and as such only the case of columns will be discussed.

The heuristic proposed involves assigning a number to a partition and then try to place into that partition all vectors that are structurally orthogonal with vectors already in the partition. If a vector cannot be placed in a partition, then it will be attempted to place it in another partition, with the next higher number. These partitions form a seed matrix. This allows the seed matrix  $S \in \{0, 1\}^{n \times q}$  to be used, where  $q \ll n$ .

*Structural orthogonality* Two columns of a matrix are structurally orthogonal if and only if in each row of both columns there exists at most one nonzero element. In Figure 2.4 the columns 1 and 2 of  $J$  are structurally orthogonal while the columns 1 and 4 are not because both have a nonzero element in the first row.

Seeding can be viewed as having three possible methods to achieve the goal of improving performance: a column-based compression, a row-based compression, or a column and row based compression. The first two are very similar, given row-based compression is the same as column-based compression on the transposed matrix. The only restriction to this partitioning is that the vectors in any such partition must be orthogonal to each other.

Let  $\tilde{J}$  be the compressed Jacobian matrix. This matrix is obtained by computing the partial derivative for each seed vector. By computing the compressed matrix one can use fewer runs than the total number of columns / rows of the original  $J$ . The seed matrix  $S$  used to determine the seed vectors can be constructed by looking at which elements of the Jacobian are nonzero and then trying to compress  $J$ . Therefore, I will always employ the sparsity structure of the Jacobian to compute the structure of the seed matrix.

$$\begin{array}{c}
 \begin{pmatrix} \tilde{j}_{1,1} & \tilde{j}_{1,4} & \tilde{j}_{1,5} \\ \tilde{j}_{2,2} & \tilde{j}_{2,3} & \\ \tilde{j}_{3,1} & \tilde{j}_{3,3} & \\ & \tilde{j}_{4,4} & \\ & & \tilde{j}_{5,5} \end{pmatrix} = \begin{pmatrix} j_{1,1} & & & j_{1,4} & j_{1,5} \\ & j_{2,2} & j_{2,3} & & \\ j_{3,1} & & j_{3,3} & & \\ & & & j_{4,4} & \\ & & & & j_{5,5} \end{pmatrix} \begin{pmatrix} 1 & & & & \\ & 1 & & & \\ * & & 1 & & \\ & & & 1 & \\ & & & & 1 \end{pmatrix} \\
 \tilde{J} \qquad \qquad \qquad J \qquad \qquad \qquad S
 \end{array}$$

Figure 2.4: The vectors are chosen in such a way that  $J$  has no nonzero elements on the same row in this column-wise compression.  $S$  is composed of the new partial derivation vectors. The resulting compressed matrix is  $\tilde{J}$ .

*Steps in computing a Jacobian* The following steps must be undertaken to determine the Jacobi matrix  $J$  with the help of seeding techniques.

1. Determine the seed matrix  $S$  by using the known sparsity structure of  $J$
2. Compute  $\tilde{J}$  using the seed matrix  $S$  in  $q$  steps
3. Extract  $J$  out of  $\tilde{J}$  with the information from  $S$

There are multiple ways to approach this, the direct method, the substitution method and the elimination method among them. This work focuses on the direct method.

*Direct method* The direct method implies that after determining  $\tilde{J}$  each element can be immediately (directly) obtained. This implies that no two columns can be computed in the same step if they both have a nonzero in the same row.

*Partitioning* The process of creating groups of columns or rows is called partitioning. Each such group can be computed with one pass of the AD tools. The partitioning can be column based, which leads to passes of forward AD, row based, and then it entails reverse AD or both and then it will require both techniques.

*Unidirectional Problem* Given the matrix  $B$ , representing the sparsity structure of a Jacobian matrix, find a structurally orthogonal partition of the columns or rows so that the number of groups is minimal.

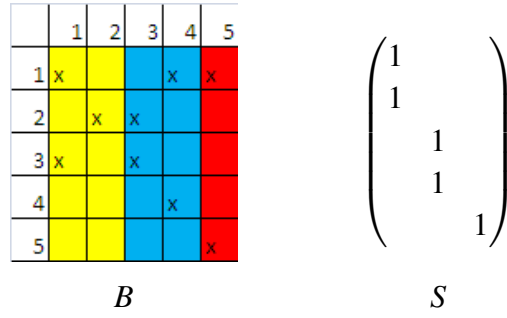


Figure 2.5:  $B$ : Column partition of a matrix  $B$  using 3 partitions. The seed matrix  $S$  has been used to determine the partitions of  $B$ . The partitions are 1, 2, 3, 4 and 5

Most of the research done so far in this field has been focused on column-based partitioning because of the wider availability of the forward mode. The row-based partitioning has been mainly used in problems which were better suited for the reverse mode.

*Bidirectional problem* Given the matrix  $B$ , representing the sparsity structure of a Jacobian matrix, find a structurally orthogonal partition of the columns and a structurally orthogonally partition of rows so that the sum of such partitions is minimal. The usage of the two seed matrices must allow the complete and immediate reconstruction of  $J$  using the seed matrices.

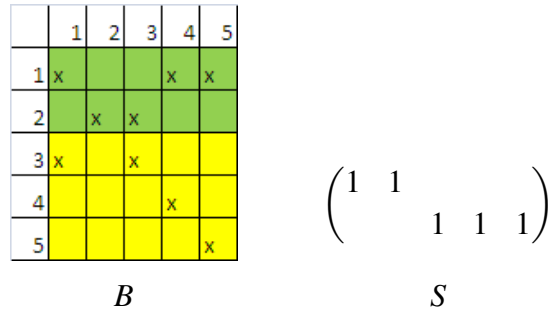


Figure 2.6:  $B$ : Row partition of a matrix  $B$  using 2 partitions. The seed matrix  $S$  has been used to determine the partitions of  $B$ . The partitions are 1, 2 and 3, 4, 5

As can be seen in the Figure 2.7, there are cases where a bidirectional partitioning is strictly better than either one of the unidirectional methods. This appears to be the case in many scientific applications as well [CV98], and since

the reverse mode has ceased to be a costly rarity for some time, bidirectional partitioning is an avenue worth exploring. In the example the heart of the issue can be observed: When a matrix has both rows and columns which are densely populated the unidirectional approach cannot be effective. The matrix has both the first row and the first column full so no compression is possible in the unidirectional approach. By using the bidirectional approach, the densely populated rows are made into partitions onto themselves and the few elements which are left are compressed as usual. This kind of pattern appears often in sparse matrices, as will be seen later in the samples taken from Tim Davies matrix collection.

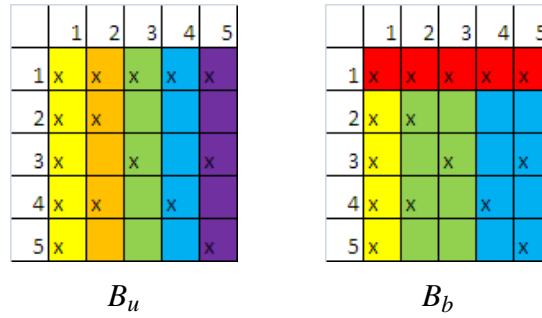


Figure 2.7:  $B_u$ : Unidirectional partitioning of  $B$ . Notice that 5 colors are needed.  $B_b$ : Bidirectional partitioning of  $B$ . Notice that only 4 colors are needed. The partitions are 1 for rows and 1, 2, 3 and 4, 5 for columns

## 2.3 Graph coloring

*Motivation* Partitioning can be modeled as a graph coloring problem. By representing the matrix as a bipartite graph  $G$ , as shown in Figure 2.10, and then coloring the bipartite graph. Before going into details about the bipartite graph some introductory notes into graph theory will be presented.

### 2.3.1 Preliminary concepts and notions

Hereby are presented introductive notions in graph theory [GMP05] which are essential to the understanding of the graph coloring algorithms presented in this work.

*Graph* A graph  $G$  is an ordered pair  $(V, E)$ ,  $V$  being a non-empty, finite set of vertices and  $E$ , the edges, being a set of unordered pairs of distinct vertices of the form  $(v_1, v_2)$ , where  $v_1, v_2 \in V$ .

*Adjacent* If the vertices  $v_1, v_2 \in V$ , and the pair  $(v_1, v_2) \in E$  then  $v_1$  and  $v_2$  are called adjacent.

*Path* A succession of adjacent vertices forms a path. The number of edges in the path gives the length of the path.

*Distance- $k$  neighbor* Two vertices are called distance- $k$  neighbors if there is a shortest path of length at most  $k$  connecting them.

*Degree- $k$  of a vertex* The number of distinct edges incident to a vertex. The degree- $k$  is the number of distinct paths of length at most  $k$  incident to that vertex. A shorthand notation for degree-1 used in this work is degree.

*Cover* A cover of a graph is a subset of vertices, so that all vertices of the graph have at least one incident edge to a vertex in the cover. The graph induced by the cover is the graph comprised of the vertices of the cover and the edges that have both endpoints in this set.

*Power of a graph* The graph  $G^k$  is the  $k$ -th power of a graph if it contains edges between all vertices that are distance  $k$ -neighbors.

### 2.3.2 Graph coloring

*Definition* Graph coloring is a well known problem in algorithmics and computer science in general. It represents the following problem: Given a graph  $G = (V, E)$ , with  $v_1, \dots, v_n$  vertices find a way to assign each node  $v_i$  in the graph a color so that no adjacent node has the same color assigned. In short, find a combination  $c_i, c_j \in \mathbf{N} \quad i, j \in 1, \dots, n$  of colors  $c_i, c_j$  are assigned to  $v_i, v_j$  so that  $c_i \neq c_j$ , for  $(v_i, v_j) \in E$

A trivial solution is to give each node a unique color but usually the goal is to minimize the number of color used

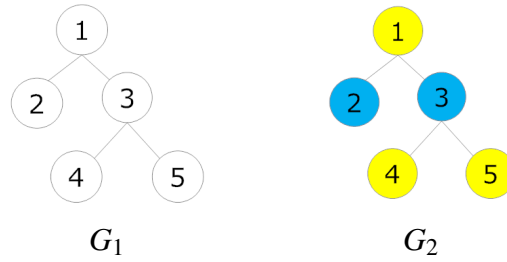


Figure 2.8:  $G_1$ , an uncolored graph and  $G_2$  a colored graph, using a distance-1 coloring. In this example the top node was colored first followed by the next row and finally the bottom row. Other coloring schemes are possible, optimal and otherwise.

*Coloring scheme* A set of colors assigned to all vertices of a graph is a coloring scheme. To be valid it must enforce certain properties depending on the type of coloring attempted (such as no adjacent nodes have the same color assigned in the case of classical coloring).

*Distance- $k$  coloring* Obtaining a coloring scheme so that no distance- $k$  neighbors of any given vertex have the same color as that vertex is called a valid distance- $k$  coloring. The number of colors needed increases due to the additional restrictions imposed by having more than the immediate neighbors colors as forbidden.

*Chromatic number* The smallest number of colors which still allow a valid coloring scheme is the chromatic number of that graph. A lot of work has been done over the years in determining efficient distance-1 coloring algorithms. The problem is that not as much is known about distance-2 coloring, and even less about how to color a bipartite graph efficiently while respecting the conditions enforced by the fact that the graph represents a Jacobian matrix. The graph in figure 2.8 has a chromatic number of 2. If the graph in the example above would have an edge connecting the node on the top with any node on the bottom than it would be impossible to color with just two colors. Thus, it would have a chromatic number of 3.

Trying to find coloring schemes that are close to the chromatic number is the goal of most graph coloring problem formulations.

### 2.3.3 Bipartite graphs

#### Bipartite graph theory

*Definition* A graph is bipartite if the vertices can be split into two disjoint sets so that no edge connects vertices in the same set. Another formulation is that a bipartite graph is a graph that contains no odd length cycles.

#### Matrix representation as a bipartite graph

The partitioning problem of the Jacobian can be formulated as a graph coloring problem for a bipartite graph. Thus, it is important to know how to represent a sparse Jacobian matrix as a bipartite graph. Let us consider the matrix in the Figure 2.9. First, one creates a node in the graph for each row and each column in the matrix. Edges between nodes represent nonzero elements and connect the row node and the column node corresponding to the element's position in the matrix. This ensures that there will be no edges between either rows or columns. The result is that the rows and columns nodes form independent sets thus creating a bipartite graph. An example can be seen in the figure 2.9. The element at 1,3 is represented by the edge form  $R1$  to  $C3$ . The bipartite nature of the graph lies in the fact that no node labeled  $R$  has edges to other  $R$  nodes, and the same is true for  $C$  labeled nodes.

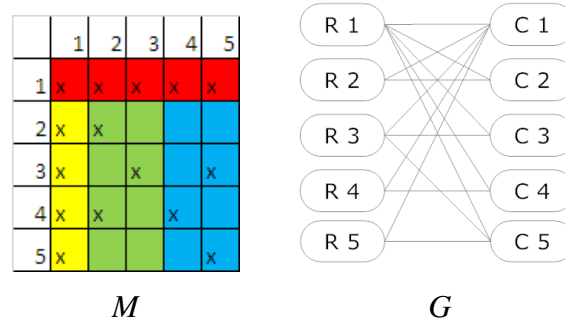


Figure 2.9: Representation of a matrix  $M$  as a bipartite graph  $G$  by creating nodes for rows and columns and then adding edges between rows and column nodes for each nonzero element in the matrix.

#### Bipartite graph coloring

Having determined the bipartite graph representation of a Jacobian matrix, the goal is to color the graph in such a way as to obtain correct and if possible optimal partitions. The conditions that have to be met when considering the



coloring depend on what kind of partitioning is required, and will be explained in the following.

*Unidirectional computation* The simplest coloring is the complete unidirectional partitioning of the matrix. This involves a distance-2 coloring of the rows or columns. For more details about the reasons for such a partitioning see Section 2.6

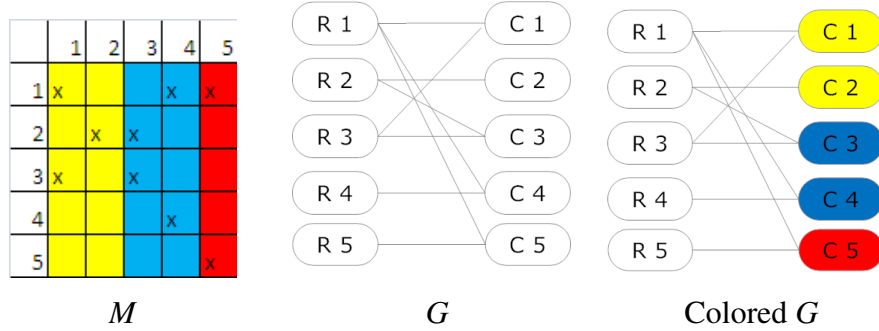


Figure 2.10: Bipartite graph and unidirectional coloring of a matrix. The condition that must be upheld is that no two nodes that share a neighbor can have the same color.

### Bidirectional computation

There are many cases where bidirectional partitioning has vastly better results when compared to unidirectional partitioning [GMP05]. Consequently, bidirectional coloring of the graph is also an important step, although the challenges are greater. The constraints involved in this coloring do not match any classical graph coloring paradigm so has not been studied as extensive as for example the distance-2 coloring problem. Using a bidirectional coloring it is not required to compute an element by both row and column. One of the two is sufficient and as such some nodes remaining uncolored due to the fact the edge is considered using the other end node. These nodes will be assigned the color 0, a neutral color meaning "not colored". These notations will persist all throughout this thesis. A couple of conditions which are sufficient to have a correct coloring are presented, and will be used throughout this work in demonstrations and algorithms.

*Star bicoloring* [GMP05] Let  $G_b = (V_1, V_2, E)$  be a bipartite graph. A mapping  $\phi : [V_1, V_2] \rightarrow \{0, 1, \dots, p\}$  is a star bicoloring of  $G_b$  if the following conditions hold:

1. If  $u \in V_1$  and  $v \in V_2$ , then  $\phi(u) \neq \phi(v)$  or  $\phi(u) = \phi(v) = 0$
2. If  $(u, v) \in E$ , then  $\phi(u) \neq 0$  or  $\phi(v) \neq 0$
3. If vertices  $u$  and  $v$  are adjacent to a vertex  $w$  with  $\phi(w) = 0$  then  $\phi(u) \neq \phi(v)$
4. Every path on four vertices uses at least three colors.

These are the conditions enforced by all of the algorithms that will be implemented in this thesis, as I focus on this type of coloring, be it complete or restricted. The color 0 is assigned to the nodes that are not supposed to be computed at all because all their incident edges are already covered by other nodes. It might pay off to analyze the coloring of nodes with this color more attentively as these choices translate into potentially fewer calculations in the end. The following property can be deduced from the above constraints. Let  $G_b = (V_1, V_2, E)$  be a bipartite graph and  $\phi : [V_1, V_2] \rightarrow \{0, 1, \dots, p\}$  a star bicoloring of  $G_b$ . Then

- the set  $C = \{v \in V_1 \cup V_2 : \phi(v) \neq 0\}$  is a vertex cover of  $G_b$
- the set  $I = \{u \in V_1 \cup V_2 : \phi(u) = 0\}$  is an independent set of  $G_b$

Obviously,  $|I| + |C| = |V_1| + |V_2|$ .

A larger independent set is desired, as these elements need not be computed. This however is **not** the actual priority, but rather choosing the independent set in such a way that the vertex cover can be colored with as few colors as possible. In [CV98] more suggestions on how the vertex cover should be chosen exist, and also indications that an optimal cover will contain vertices of both  $V_1$  and  $V_2$ . This seems to be particularly the case for such vertices that have a high number of distance-1 neighbors even if this means the cover will not be minimal.

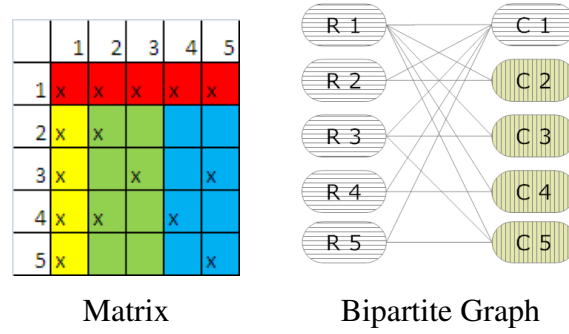


Figure 2.11: Bipartite graph resulting of a matrix. The vertex cover is represented with horizontal hashing while the independent set is represented with a vertical hashing.

The scheme of the coloring algorithm from which I have started my work and which represents the state of the art in this field to the best of my knowledge, is this:

---

**Algorithm 2.3.1** Scheme for star bicoloring

---

**procedure** STARBICOLORINGSCHEME( $G_b = (V_1, V_2, E)$ )

1. Find a suitable vertex cover  $C$  in  $G_b$
2. Assign the vertices in the set  $I = (V_1 \cup V_2) \setminus C$  the color 0
3. Color the vertices in  $C$  such that the result is a star bicoloring of  $G_b$

**end procedure**

---

### 2.3.4 Graph coloring framework

I have used the framework provided by Michael L  lfesmann in this thesis. This constitutes in programs which take in the parameters for the run and can then use numerous heuristics on a given matrix and allow for customizing the tests, choosing the value of  $\rho$ , using a particular kind of preordering, or none at all. This has allowed me to get right to developing my own algorithms, after a short while adjusting to the project specifics.

### 2.3.5 Testing

The project has been tested using matrices from the Tim Davies Sparse Matrix Collection. They have been chosen so that the results can be compared to the previous state of the art implementations of graph coloring algorithms for sparse

Matrix	#Columns	#Nonzeros	Symmetrical	Sparsity (in %)
Ge87H76	112,985	7,892,195	1	0.0618
barrier2-10	115,625	3,897,557	0	0.0292
bmwcra_1	148,770	10,644,002	1	0.0481
c-73	169,422	1,279,274	1	0.0045
cage12	130,228	2,032,536	0	0.0120
cont-300	180,895	988,195	1	0.0030
d_pretok	182,730	1,641,672	1	0.0049
epb3	84,617	463,625	0	0.0065
ford2	100,196	544,688	1	0.0054
hcircuit	105,676	513,072	0	0.0046
lung2	109,460	492,564	0	0.0041
matrix_9	103,430	2,121,550	0	0.0198
ohne2	181,343	11,063,545	0	0.0336
para-5	155,924	5,416,358	0	0.0223
pkustk13	94,893	6,616,827	1	0.0735
rajat23	110,355	556,938	0	0.0046
scircuit	170,998	958,936	0	0.0033
ship_003	121,728	8,086,034	1	0.0546
shipsec1	140,874	7,813,404	1	0.0394
stomach	213,360	3,021,648	0	0.0066
torso2	115,967	1,033,473	0	0.0077
turon_m	189,924	1,690,876	1	0.0047
twotone	120,750	1,224,224	0	0.0084
xenon2	157,464	3,866,688	0	0.0156
$\Sigma =$	3,317,614	83,859,881		

Table 2.2: 24 Matrices from the Tim Davis Matrix Collection with their major characteristics: number of columns, number of nonzeros, symmetry and sparsity

Jacobian computation.

The matrices are representative examples of sparse matrices used in scientific computing and thus are a prime candidate for testing the algorithms. Testing time is in the hours range for most matrices due to their great size.

The structure of the matrices is diverse but a few can be divided into categories:

1. *Diagonal-dense Matrices* - matrices with very few non-zero elements outside the main diagonal, or a close proximity to the main diagonal.
2. *Block Matrices* - matrices where a particular structure appears repeatedly, possibly slightly altered.

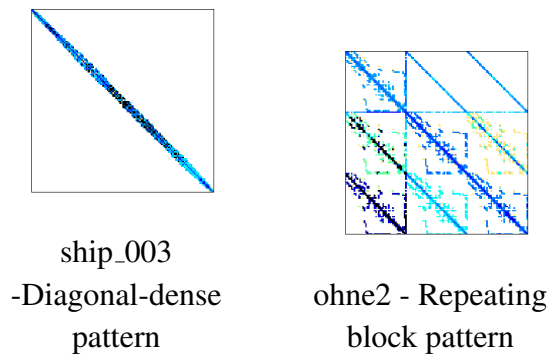


Figure 2.12: Examples of matrices from the Tim Davies Collection

### 2.3.6 Challenges

The greatest challenge presented by this thesis was to discover improvements to the algorithms provided. My goal was not to take existing code and simply improve it, but rather find conceptual improvements. Temporal and spacial restrictions are not the issue, as the run time and memory space needed by the graph coloring program are usually smaller than even one run of the AD tool. As such, if the coloring algorithm takes twice as much time and improves the number of colors by just one it would still be a gain.

## 2.4 Algorithm for graph coloring - overview of issues

Research for this project has allowed some concepts to emerge as being highly relevant in the heuristics used and improving them would have a high probability of bettering the performance of the algorithm.

### Strictness of the conditions

A problem faced by any programmer is how to implement the conditions of a problem as near to their definition as possible. In this case, the question is, how do you color the nodes so that you end up with the optimal number of colors (or at the very least something close). Since the problem of graph coloring is NP-complete [Lül06], heuristics must be used. Finding a way to implement the conditions as close to their idea as possible and still obtain a valid coloring could improve the algorithm.

## Ordering of vertices

The order in which the vertices are colored is essential to the performance of the program. The basic observation [CGM84] from which most heuristics targeting the order of vertices emerge from is that coloring vertices with a high number of distance-1 neighbors should be done before those with fewer neighbors, as more neighbors equate to more restrictions, and thus an increased chance of having to use a new color. While this is by no means true for all graphs, this patterns emerges often enough to be considered the common case.

A solution has been suggested in [HS98]. This entails the preordering of the nodes, before even the independent set is chosen, so that high-degree vertices are colored first. These are the orderings suggested:

- LFO: Largest first ordering: for the ordering  $v_1, \dots, v_i, \dots, v_j, \dots, v_n$ , the property  $\text{degree}(v_i) \leq \dots \leq \text{degree}(v_j)$  holds
- SLO: Smallest last ordering: the  $k^{\text{th}}$  vertex  $v_k$  is determined after  $v_{k+1}, \dots, v_n$  have been selected by choosing  $v_k$  so that its degree in the subgraph induced by  $V / \{v_{k+1}, \dots, v_n\}$  is minimal.
- IDO: Incidence degree ordering: the  $k^{\text{th}}$  vertex  $v_k$  is determined after  $v_1, \dots, v_{k-1}$  have been selected by choosing  $v_k$  so that its degree in the subgraph induced by  $V / \{v_1, \dots, v_{k-1}\}$  is maximal.
- Dynamic LFO: Dynamic largest first ordering: Similar to LFO, but each vertex is removed from the graph after it has been placed in the ordering.

The advantage of this is that high degree nodes get to be colored first. It is possible effectively fill the partitions by "filling in the gaps" with lower degree nodes. This way, closer to the end of the coloring only small degree nodes should be left over and these should be easier to color. Usage of these orderings brings good results as opposed to using no ordering whatsoever as can be observed in [Lül06], but my opinion is more could be gained if this ordering would be done dynamically in the course of the coloring. The results in the following Table 2.3 are proof of the improvements brought about by the use of preorderings. The matrices used are from Tim Davies collection and will be presented in more detail in the next chapter.

<i>Star Bicoloring Algorithm</i>	–	<i>LFO</i>	<i>SLO</i>	<i>IDO</i>
barrier2-10	118	118	118	118
bmwcra_1	186	195	138	141
c-73	57	57	57	59
cage12	96	72	68	70
cont-300	12	10	9	10
d_pretok	22	21	20	20
epb3	8	7	7	6
ford2	38	33	33	33
hcircuit	23	21	21	20
lung2	8	14	8	8
ohne2	204	204	204	204
para-5	162	162	162	162
pkustk13	303	300	300	300
rajat23	102	100	100	100
scircuit	86	85	85	85
ship_003	181	168	144	144
shipsec1	126	126	114	114
stomach	38	36	30	32
torso2	18	13	12	13
turon_m	23	22	20	20
twotone	105	84	87	89
xenon2	52	51	41	43
$\Sigma =$	1.968 32	1.899	1.779	1.785

Table 2.3: Number of colors obtained with the *StarBicoloringSchema* without preordering and with preordering *LFO*, *SLO* and *IDO*

## 2.5 Partial Jacobian computation

### 2.5.1 Motivation

*Solving linear equations* In many cases for solving of a system of linear equations preconditioners are used. These increase the convergence of the iterative solution of the equation system and thus allow for a faster determination of the system. The preconditioner must be chosen so that the total time required does not become larger. Let us assume we use the preconditioner  $M$ , a  $\mathbf{R}^{n \times n}$ .

$$M^{-1}Az = M^{-1}b, \quad \text{(left preconditioner)}$$

$$AM^{-1}y = b \quad \wedge \quad z = M^{-1}y \text{ and} \quad \text{(right preconditioner)}$$

$$M_L^{-1}AM_R^{-1}y = M_L^{-1}b \quad \wedge \quad z = M_R^{-1}y, \quad M = M_L \cdot M_R, \quad \text{(two-sided preconditioner)}$$

To be able to take advantage of a preconditioner additional steps must be taken. First the matrix  $M$  must be computed, second the system  $Mu = v$  solved and/or the product  $z = Mv$  determined.

When choosing  $M$  the two extremes must be balanced: convergence should be achieved as quickly as possible but without spending too much time on computations. The extremes in question are  $M = I$ , the identity matrix, which means no additional computations required, but also no change to the system and  $M = A$ , requires the full computation of the inverse of  $A$ , but provides convergence in one step.

Up to now all elements of the Jacobian matrix were computed. With the use of a Jacobi preconditioner it is possible to need just a subset of elements. In other words if one uses a preconditioner it is possible to determine the elements of the Jacobian that have to be computed and those elements that, in while not zero are not required.

### 2.5.2 Concepts

As seen above, often in practice not all values of the Jacobian are needed, thus, some elements of the matrix, though nonzero are of no interest. This means that they can be ignored when creating partitions as long as these elements do not affect the computation of required elements. This means that rows or columns with nonzero elements on the same position can be compressed as long as these elements are not needed. This observation is important to bidirectional computation as well, given the fact that nodes need be computed just once, either by column or row. This implies that elements already computed on rows can be ignored when computing columns and vice versa.

In the Figure 2.13 the first and fourth column can be placed into the same partition because the only nonzero elements on the same row are not among those required.



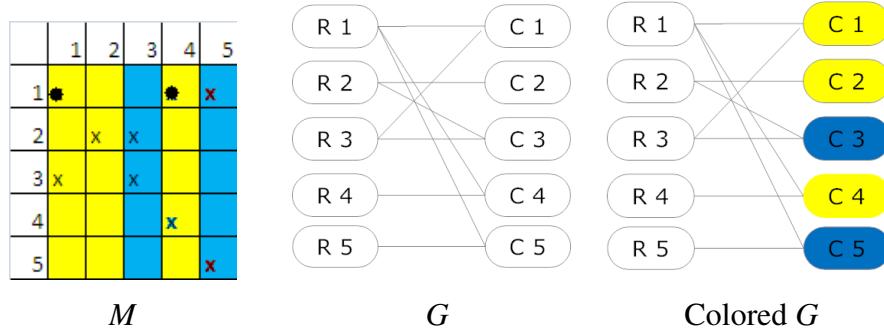


Figure 2.13: Bipartite graph  $G$  and restricted unidirectional coloring of a matrix  $M$ . The blacked out elements in the matrix are those which are of no interest. Notice that a fewer number of colors are needed in comparison to the example in Figure 2.10

**Results** As can be seen in the following comparison, restricted computation can be a good way to reduce the number of passes required to obtain the required elements of a Jacobian. The Table 2.4 represents the summation of Michael Lüllesmanns work in [Lül06] and the dramatic improvements obtained by using bidirectional coloring and restricted coloring respectively.

Coloring Algorithm	Number of colors (total)
<i>UnidirectionalD2Coloring</i>	70.207
<i>StarBicoloringSchema</i> ( $p = 3, 0$ )	2.694
<i>StarBicoloringSchema</i> ( $p = 3, 0$ ) + <i>SLO</i>	2.435
<i>RestrictedUnidirectionalD2Coloring</i>	655
<i>RestrictedUnidirectionalD2Coloring</i> + <i>SLO</i>	586

Table 2.4: Sum of colors obtained by the various algorithms

### Restricted bidirectional computation

Similarly to the restricted unidirectional computation, not having to actually compute all elements can bring significant advantages in performance. Depending on the structure of required elements, it is possible that the bidirectional coloring will not bring better results than the unidirectional coloring.

In [Lül06] it has been pointed out that for a Jacobi preconditioner (the required elements are those on the main diagonal of the matrix) the bidirectional restricted computation brings nothing in comparison to the unidirectional computation, as can be seen in this Table 2.5: Thus, in the domain of bidirectional restricted

Matrix	RestrictedUnidirectionalD2Coloring $V_c$	$V_r$	RestrictedStarBicoloringSchema $\rho = 1, 0$
Ge87H76	224	224	224
barrier2-10	26	26	26
bmwcra_1	40	40	40
c-73	2	2	2
cage12	14	14	14
cont-300	2	2	1
d_pretok	6	6	6
epb3	5	5	5
ford2	28	28	28
hcircuit	5	5	5
lung2	4	4	4
matrix_9	7	7	7
ohne2	36	36	36
para-5	27	27	27
pkustk13	57	57	57
rajat23	9	9	9
scircuit	10	10	10
ship_003	60	60	60
shipsec1	48	48	48
stomach	8	8	8
torso2	5	5	5
turon_m	7	7	7
twotone	5	5	5
xenon2	20	20	20
$\Sigma =$	655	655	654

Table 2.5: Number of colors (without the color 0) obtained by uni- and bidirectional restricted computation algorithms

computation two challenges emerge.

1. Try to improve the existing algorithm so that it needs a lower number of colors
2. Find other patterns that can take advantage of the restricted bicoloring algorithm

I will first attempt to improve the state of the art algorithm with the same techniques developed in this work for the complete bicoloring method, and only then move on to discover new patterns. The theoretical research done so far in this field is quit limited and as such improvements can discoveries can still be made.

## Chapter 3

---

# Hardware and Software Configuration

---

All hardware and software used in the development and writing of this work has been offered by the Institute for Scientific Computing of RWTH Aachen University in Germany and by S.I. Dr. Ing. Emil Slușanschi from the Politehnica University of Bucharest. All source code written by me during this time will remain at the institute as well as at the Politehnica University.

### 3.1 Hardware

I have been granted the opportunity to write my bachelor thesis at the Institute for Scientific Computing of RWTH Aachen University in Germany. As such I was able to utilize the infrastructure here to evaluate, test and improve my project and its constituent algorithms. Although I could have handled the development on my laptop, I could never run the extensive batteries of test used without using the cluster of the institute. For development I have used a workstation with cluster access provided by the Institute for Scientific Computing and my own laptop.

Code developed was kept in source depot using subversion (SVN) for backup and history reasons. This has also allowed for easier code reviews and bug tracking.

For testing I have used the Linux cluster of the Institute, described in the following figure 3.1. The cluster had two ways of submitting jobs, a direct one where the user would log onto the cluster and from there simply run the programs needed and a batch system. The direct approach was acceptable only for short jobs, as the maximum time allowed for running jobs in this manner was 20 minutes and was used mainly for testing and compiling. For longer computations the batch system was preferred.

The batch system permits users to queue any number of jobs and specify for each the desired architecture, maximum running time, space requirements and other parameters. Combining the power and flexibility of the batch system with bash scripts I could run my tests efficiently and without wasting additional time in gathering test results.

There are two ways to run programs in the batch queue, one in the general system, which allows it to look in all the clusters available and try to find a free spot, or one in which one identifies himself as a member of a research group and receives a higher priority in the groups own cluster. There are advantages and disadvantages to each option, as the first allows for a much larger pool of machines on which to run tests, although with a low priority whilst the second allows for a high priority on the local cluster but a limited choice of machines. The second alternative is usually preferred, but on occasion the local cluster can be so swamped with jobs that a high priority will not mean a quick result.

The priority system is built to allow programs that run for a short time, and on few processors to run before the others, and as such I have adapted and tried to have my test suites as modular as possible. Where in the beginning I had one great script that did all the tests, all the checking, in the meantime it was evolved into a suite of batch scripts allowing partial runs, with configurable parameters.



















X86-64 basiert	Anzahl Systeme	Betriebs-system <sup>1)</sup>	Prozessor-typ	Prozessoren/ Kerne	Speicher <sup>2)</sup>	Namen <sup>1)</sup>
Fujitsu-Siemens RX600 S4/X <sup>3)</sup>	2	 	Xeon X7350, 2,93GHz (quad core)	4/16	64 GByte	linuxhtc01-02 winhtc01-02
Fujitsu-Siemens RX200 S4/X <sup>3)</sup>	10	 	Xeon E5450, 3,0GHz (quad core)	2/8	32 GByte	linuxhtc03-12 winhtc03-12
Fujitsu-Siemens RX200 S4/X <sup>3)</sup>	50	 	Xeon E5450, 3,0GHz (quad core)	2/8	16 GByte	linuxhtc13-62 winhtc13-62
Dell 1950	4	 	Xeon 2,66GHz, (quad core)	2/8	16 GByte	linuxctc00-03 winctc00-03
Dell 1950	7	 	Xeon 3,0GHz, (dual core)	2/4	8 GByte	linuxwcc01-07 winwcc01-07
Dell 1950	2	 	Xeon 3,0GHz, (dual core)	2/4	16 GByte	linuxwcc0 winwcc00
Sun Fire V40z	64	 	Opteron 848, 2,2GHz	4/4	8 GByte	linuxoc0-63 winoc00-63 sunoc00-63
Sun Fire V40z	4	 	Opteron 875, 2,2GHz (dual core)	4/8	16 GByte	linuxoc64-67 sunoc64-67
Sun Fire X4600	2	 	Opteron 885, 2,6GHz (dual core)	8/16	32 GByte	linuxoc68-69 sunoc68-69

Figure 3.1: Cluster information from the RZ homepage

## 3.2 Software

This thesis has been completed using only free and licensed software. For the implementation part of the project the following programs have been used:

- Free: g++, totalview, kate, STL, boost, matrixmarket
- Licensed: Matlab

For the writing of this thesis the following programs have been used:

- Free: LaTeX(TeXNicCenter, MikTex), Graphviz, uDraw, SmartDraw
- Licensed: Microsoft Office

## 3.3 Project implementation

### 3.3.1 Preliminary issues

The project is the continuation of the work of Michael Lülkesmann, and as such is based on his work. I have been provided with the source code of this work and have built upon the framework he has provided. The language used is C++, and a lot of the testing has been automated using bash scripting. To ensure the correctness of the algorithms I have written a "checker", a program that receives as input the output of the graph coloring algorithm and the subsequently

checks the four conditions for a correct star bicoloring as defined in [GMP05]. To generate some smaller examples of sparse Jacobian matrices I have also written a generator program who is able to analyze input parameters and generate any size of nonsingular matrices with any given element density.

## Chapter 4

---

# Two-sided coloring algorithms for complete Jacobian matrix computation

---

This chapter describes existing algorithms for star coloring used as a starting point as well as those developed while working on the thesis. In this chapter part each algorithm has been tested for correctness with the checker tool mentioned in the previous chapter and has attached the results obtained. First a short description of some of used terms:

- *Concept* A short description of the heuristic and why it was considered worthy of implementation.
- *Algorithm* The algorithm in pseudocode.
- *Results* The table of results for various tests.
- *Discussion* An explanation as to the nature of the results, as well as the path that has lead to the discovery of said explanation.
- *Conclusion* Discussion of the results and of the value of the heuristic.

The source code, written in C, is not included here. Instead a pseudocode form is offered. Samples of the source code can be found in the appendix, and the complete source code can be accessed as part on the digital format attached.

## 4.1 Star bicoloring scheme

### Concept

This algorithm is presented in [GMP05]. The basis is an implementation in [Lül06] and as such is the starting point for my work. It represents the state of the art in the field of two-sided graph coloring for sparse Jacobian computation. I will introduce it here as a reference point to all following algorithms.

### Algorithm

---

**Algorithm 4.1.1** Star Bicoloring Algorithm for  $G_b$ 


---

```

procedure STARBICOLORING( $G_b = (V_r, V_c, E), Color$ )
  Let  $v_1, v_2, \dots, v_{|V_r \cup V_c|}$  a given ordering of  $V_r \cup V_c$ 
  Initialize Array forbiddenColors with 0
  for  $i := 1, \dots, |V_r \cup V_c|$  do
    if  $Color[v_i] \neq 0$  then
      for each  $w \in NeighborDistance1(v_i, G_b)$  do
        if  $Color[w] \leq 0$  then
          for each  $x \in NeighborDistance1(w, G_b)$  with  $Color[x] > 0$  do
            forbiddenColors[ $Color[x]$ ] :=  $i$ 
          end for
        else
          for each  $x \in NeighborDistance1(w, G_b)$  with  $Color[x] > 0$  do
            for each  $y \in NeighborDistance1(x, G_b)$  with  $Color[y] > 0$ 
              and  $y \neq w$  do
              if  $Color[w] = Color[y]$  then
                forbiddenColors[ $Color[x]$ ] :=  $i$ 
              end if
            end for
          end for
        end if
      end for
       $Color[v_i] := \min\{j > 0: forbiddenColors[j] \neq i\}$ 
    end if
  end for
  Add  $\max\{Color[v_r] : v_r \in V_r\}$  to all  $Color[v_c]$  with  $v_c \in V_c$  and  $Color[v_c] > 0$ 
  return Color
end procedure

```

---

The algorithm begins after the independent set presented in Chapter 2 2.11 has been determined, so that all the nodes that need not be computed already have



assigned the nocolor 0. The restrictions that allow for a consistent star bicoloring will be listed here again for ease of use:

*Star bicoloring* [GMP05] Let  $G_b = (V_1, V_2, E)$  be a bipartite graph. A mapping  $\phi : [V_1, V_2] \rightarrow \{0, 1, \dots, p\}$  is a star bicoloring of  $G_b$  if the following conditions hold:

1. If  $u \in V_1$  and  $v \in V_2$ , then  $\phi(u) \neq \phi(v)$  or  $\phi(u) = \phi(v) = 0$
2. If  $(u, v) \in E$ , then  $\phi(u) \neq 0$  or  $\phi(v) \neq 0$
3. If vertices  $u$  and  $v$  are adjacent to a vertex  $w$  with  $\phi(w) = 0$  then  $\phi(u) \neq \phi(v)$
4. Every path on four vertices uses at least three colors.

The idea of the algorithm is to take each vertex in turn and examine its distance-1, -2 and -3 neighbors and mark each color that cannot be assigned to the node because of the restrictions.

After this step is completed, a color is assigned to the node by choosing the color with the smallest value that has not been marked as forbidden.

The conditions that mark a color as forbidden are a simple reflection of the restrictions from [GMP05] and will be discussed in the following.

## Discussion

This is the best Algorithm 4.1.1 up to date, and its implementation is quite close to the initial restrictions of the coloring.

1. If  $u \in V_1$  and  $v \in V_2$ , then  $\phi(u) \neq \phi(v)$  or  $\phi(u) = \phi(v) = 0$
2. If  $(u, v) \in E$ , then  $\phi(u) \neq 0$  or  $\phi(v) \neq 0$
3. If vertices  $u$  and  $v$  are adjacent to a vertex  $w$  with  $\phi(w) = 0$  then  $\phi(u) \neq \phi(v)$
4. Every path on four vertices uses at least three colors.

The first condition is automatically enforced by having separate numbers to color rows and columns, meaning the color "1" in a row is not the same as color

”1” in a column. The second condition is enforced by not permitting the color 0 as a valid color after the independent set computation. The third condition is enforced by the  $Color[w] \leq 0$  branch of the *if*, while the fourth is enforced by the *else* branch.

This leads to a very clean and elegant implementation which allows preordering and constitutes a stable and expandable base from which to start development.

## 4.2 Complete direct cover

### Concept

This implementation suggested by [HS98] is a different approach to bicoloring algorithms. The complete direct cover (CDC) involves the direct coloring of the graph without first computing the independent set. A cover is determined and the elements left at the end of this process not included in the cover form the independent set.

For the coloring itself the method suggested is to color as many nodes as possible with one color before moving on to the next. This would equate to a program that tries to ”fill” the rows/columns in the compressed Jacobian. The implementation uses LFO to try to use nodes with a higher degree first so that the easier to color low degree nodes remain last.

## Algorithm

---

**Algorithm 4.2.1** Complete direct cover heuristic for bidirectional (2- sided) coloring

---

```

procedure CDC( $G_b = (V_1, V_2, E)$ )
   $k = 0$ 
   $edgcount = 0$ 
  while  $edgcount < |E|$  do
     $k = k + 1$ 
    Let  $V_1^k \subset V_1$  and  $V_2^k \subset V_2$  be uncolored vertices
    Sort the vertices of  $G[V_1^k \cup V_2^k]$  in nonincreasing degree in  $G[V_1^k \cup V_2^k]$ 
    Let the order be defined by  $w_i, i = 1, 2, \dots, |V_1^k \cup V_2^k|$ 
     $W^k = \{w_1\}$ 
    for  $i = 2, 3, \dots, |V_1^k \cup V_2^k|$  do
      if  $w_i$  and  $w_1$  are in the same bi-partition
      and there is no  $w \in W^k$  such that  $w_i$  is connected to  $w$  by a path in  $G[V_1^k \cup V_2^k]$ 
      of length=2 then
        Add  $w_i$  to the set  $W^k$ 
      end if
    end for
    Assign the color  $k$  to every vertex in  $W^k$ 
    Let  $E_k$  be the set of edges which has no end points in  $W^k$ 
     $edgcount = edgcount + |E_k|$ 
  end while
end procedure

```

---

The algorithm takes a color and tries to assign that to as many nodes as possible. The nodes are ordered using a non-increasing degree ordering (LFO). The coloring is not of the entire graph, but rather it creates a cover of the graph and colors the vertex at the same time. Vertices left uncovered at the end of the algorithm are colored with 0 as they are part of the independent set.

## Results

The results are presented in Table 4.1, along with a comparison with the basic star bicoloring scheme. The notations on the table are as follows:

- SBS - star bicoloring algorithm
- CDC - complete direct cover

- Delta (%) Difference between SBA and CDC, in percents. A higher value represents a poorer performance

A line in the table represents the CDC algorithm taking too much time to run.

Matrix	#Colors SBA	#Colors CDC	% CDC/SBA
barrier2-10	118	-	-
bmwcra_1	186	2035	1094%
c-73	57	-	-
cage12	96	819	853%
cont-300	12	12	100%
d_pretok	22	98	445%
epb3	8	36	450%
ford2	38	321	844%
hcircuit	23	-	-
lung2	8	20	250%
ohne2	204	-	-
para-5	162	-	-
pkustk13	303	2167	715%
rajat23	102	965	946%
scircuit	86	-	-
ship_003	181	2105	1162%
shipsec1	126	663	526%
stomach	38	239	628%
torso2	18	60	333%
turon_m	23	100	434%
twotone	105	1371	1305%
xenon2	52	257	494%
$\Sigma =$	1968	11268	-

Table 4.1: Test results for complete direct cover algorithm, star bicoloring scheme and the comparison of the results

## Discussion

The implementation suggested by [HS98] is therefore not a better solution than the already implemented solutions, as the conditions imposed on the coloring are too strict, and additional unneeded colors are used. The simplest case is that of the 3-path in the graph which has to contain at least three different colors (full demonstration in [GMP05]). While this condition is correctly enforced, any implementation of CDC, while fast, will only allow three colors on a 3-path in the graph if, and only if the shared color of the two nodes sharing a color is 0,

the non-color. This will subsequently cause more colors to be used than actually required. Experimental results show that the star bicoloring scheme, with no preordering, need only a quarter of the colors used by the CDC algorithm. The star bicoloring scheme is better with one exception, when an equal result is obtained. The reason is that the conditions for a correct coloring are not used as good as possible.

## Conclusion

Thus, further research into this kind of heuristic has been discontinued as being deemed with little to no potential. The original idea of trying to fill rows/-columns sounds interesting though to warrant continued study into this type of heuristic. An adaptation of the star bicoloring scheme to contain the concepts of CDC.

## 4.3 Row/Column Fill Bicoloring Algorithm

### Concept

This implementation is an attempt to combine the CDC approach with the star bicoloring scheme, which allows usage of preordering techniques. The advantage would be that for each color, a greedy-style heuristic is used to try to fill the row or column as completely as possible. The idea would be to pick a color, and then try to find all vertices that can be colored with that color, where the vertices are ordered using a non-increasing degree ordering (LFO). This should allow for an efficient use of colors, allowing for an optimal solution.

## Algorithm

---

### Algorithm 4.3.1 Row/column Fill

---

```

procedure RCF( $G_b = (V_1, V_2, E)$ )
  Preorder LFO( $V_1$ ) LFO( $V_2$ )
  Compute Independent Set and color it with the color 0
   $k := 0$ 
   $color := 1$ 
   $vertexcount := 0$ 
  while  $vertexcount < |V_1 \cup V_2|$  do
    for each  $v_i$  in  $V_1 \cup V_2$  that is uncolored do
      ...
      SBA coloring for  $v_i$ 
      ...
      If coloring is successful increase  $vertexcount$ 
    end for
     $color := color + 1$ 
  end while
end procedure

```

---

The algorithm takes each color in an increasing fashion and tries to color as many vertices with it as possible. The algorithm stops if all nodes are colored.

## Results

The results are not included here, as they do not bring any kind of improvement over the original algorithm. The use of preordering seems to automatically bring this improvement to the algorithm, and the restrictions added serve no purpose. There has been no difference to the values obtained for the star bicoloring algorithms used in conjunction with a LFO preordering with the new heuristic.

## Discussion

While this heuristic is not worse than what already existed, it clearly does not bring something new to this field. The explanation is that using the LFO ordering causes the nodes to be colored with the same greedy approach as the row/column fill approach. This brings the possibility that a dynamic ordering approach might yield better results whilst keeping the SBS base mostly unchanged.

## Conclusion

Further research into this kind of heuristic has been discontinued as being deemed with little to no potential. It becomes obvious that the modifications that need to be brought to the original algorithm need to focus on something previously overlooked or a new direction. This new direction should possibly be the dynamic ordering of the vertices taking into consideration the colors assigned so far.

## 4.4 Star bicoloring algorithm with restriction relaxation and queue system

### Concept

While working on figuring out improvements to the star bicoloring algorithm it was discovered that one of the restrictions, namely the one stating any the color of adjacent vertices to vertices that have not been colored should not be used, caused the program in some cases to eliminate a color as a candidate when it should not, thus causing a suboptimal coloring to emerge.

It was thus tried to find a way to relax the conditions without making the algorithm create invalid coloring schemes. This was done by modifying the condition described above in the star bicoloring algorithm.

### Algorithm

The first Algorithm 4.4.1 is the first version created, which while straight forward did not create the expected results. The modified algorithm 4.4.2 tries to alleviate the problems noticed and discussed later and get a lower count of colors.

### Discussion

When running tests using this algorithm 4.4.1 it was discovered that the new conditions were too lax, and allowed for a unidirectional coloring because of this fact. This was unacceptable, and unwilling to drop this line of research at once the decision was made to implement a queue system hoping to fix this issue. The modified algorithm 4.4.2 sports a queue in which all neighbors of a colored vertex are added, preferably in an order such as LFO.

**Algorithm 4.4.1** Star bicoloring algorithm with restriction relaxation for  $G_b$ 


---

```

procedure STARBICOLORINGRR( $G_b = (V_r, V_c, E), Color$ )
  Let  $v_1, v_2, \dots, v_{|V_r \cup V_c|}$  a given ordering of  $V_r \cup V_c$ 
  Initialize Array forbiddenColors with 0
  for  $i := 1, \dots, |V_r \cup V_c|$  do
    if  $Color[v_i] \neq 0$  then
      for each  $w \in N_1(v_i, G_b)$  do
        if  $Color[w] \leq 0$  then
          for each  $x \in N_1(w, G_b)$  with  $Color[x] > 0$  do
            for each  $y \in N_1(w, G_b)$  with  $Color[y] > 0$  do
              for each  $z \in N_1(w, G_b)$  with  $Color[z] \neq Color[x]$  do
                forbiddenColors[ $Color[x]$ ] :=  $i$ 
              end for
            end for
          end for
        end for
      else
        for each  $x \in N_1(w, G_b)$  with  $Color[x] > 0$  do
          for each  $y \in N_1(x, G_b)$  with  $Color[y] > 0$  and  $y \neq w$  do
            if  $Color[w] = Color[y]$  then
              forbiddenColors[ $Color[x]$ ] :=  $i$ 
            end if
          end for
        end for
      end if
    end for
     $Color[v_i] := \min\{j > 0 : \text{forbiddenColors}[j] \neq i\}$ 
  end if
end for
  Add  $\max\{Color[v_r] : v_r \in V_r\}$  to all  $Color[v_c]$  with  $v_c \in V_c$  and  $Color[v_c] > 0$ 
  return  $Color$ 
end procedure

```

---

## Results

The results, though better than the ones obtained by CDC were not encouraging. They were significantly weaker than the established coloring heuristics and as such an explanation should be required. The explanation is that the SBA with restriction relaxation now colors many low degree vertices before coloring the higher degree ones, as often in the test matrices the immediate neighbors of high-degree vertices are not high-degree themselves. This leads to an overall poorer performance.



---

**Algorithm 4.4.2** Star bicoloring algorithm with restriction relaxation and queue system for  $G_b$

---

```

procedure STARBICOLORINGRRQS( $G_b = (V_r, V_c, E), Color$ )
  Let  $v_1, v_2, \dots, v_{|V_r \cup V_c|}$  a given ordering of  $V_r \cup V_c$ 
  Initialize Array forbiddenColors with 0
  Add  $v_1$  uncolored vertex to the queue
  while queue not empty do
    Let  $v_i$  be the top element popped out of the queue
    ...
    Color  $v_i$  as above
    ...
    Add all uncolored neighbors of  $v_i$  to the queue in an order(LFO)
  end while
  return Color
end procedure

```

---

## Conclusions

While an interesting direction at first, practice shows that this solution is not the way to obtain improved results. The idea to try a greedy approach is not disproven however, which could imply that perhaps the idea is sound only a different implementation is required. This is a problem that might need additional research to ultimately prove or disprove its validity.

## 4.5 Refinement of results using postprocessing methods

### Concept

The star bicoloring algorithm is the basis of my work. Thus, quite some time was spent analyzing the conditions and restrictions it implements, and found there are cases where it is too restrictive and thus obtains suboptimal results.

Let us take the example in the Figure 4.1. Let us assume the node 1 is the one currently being colored. The restriction as it is implemented will check if the node 2 has been colored. If it has not been colored the color of the node 3 will be considered forbidden. The case exists were the nodes 2 and 4 end up having different colors, meaning that 1 and 3 could be colored using the same color.

The restriction in question cannot be completely eliminated though, as it will lead to poor performance on the more general cases when compared to the start-

ing algorithm. Since the improvement hinges on the algorithm using a color needlessly, a simple solution would be to come back at the end and check if a 'lower' color can be used. An example is presented in the Figure 4.1.

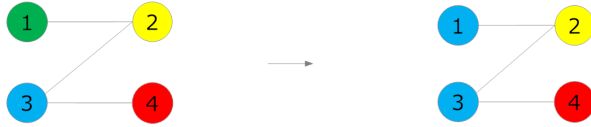


Figure 4.1: Example showing how a color can be eliminated. Note that this is just a small extract of a larger graph, only to present the concept.

## Algorithm

---

### Algorithm 4.5.1 Postprocessing

---

```

procedure POSTPROCESSING( $G_b = (V_1, V_2, E), Colors(V_1), Colors(V_2)$ )
  while Not all vertices checked do
    Let  $v$  be an unchecked vertex.
    Try to color  $v$  with a lower color.
    Check  $v$ 
  end while
end procedure

```

---

The Algorithm 4.5.1 does not bring major changes in the implementation itself but rather adds a small improvement for the particular cases mentioned above. The idea is to simply add another pass to the starting algorithm that allows it to eliminate superfluous colors.

## Results

The results are positive, but not impressive. The improvement is either nonexistent, or just one or a few colors better than the original, while the cost is quite high: A second run through all the vertices takes quite a lot of additional time, close to the original program time.

## Conclusion

This new approach is not very promising, and although the disadvantages could be limited by tagging the particular vertices that present possible opportu-

nities to eliminate their color and then checking only them, but since the advantages are so limited it is not a direction worth researching too deeply. I would consider using this refinement only if the number of colors used is critical.

## 4.6 Integrated star bicoloring scheme

### Concept

An implementation developed with the assistance of Michael Lölfesmann involves coloring the graph at the same time the independent set is constructed. The advantage comes from the fact that in this way both row- and column-vertices can be computed alternatively, rather than in the classical approach. This yields improved results to all previous implementations due to the increased adaptability. This is just a basic improvement to prove the validity of the technique and further modifications can be made to make it even more efficient.

### Algorithm

---

**Algorithm 4.6.1** Integrated star bicoloring scheme

---

```

procedure INTEGRATEDSTARBICOLORINGSCHEME( $G_b = (V_1, V_2, E)$ )
  while Vertex Cover not completed do
    Let  $v$  be the vertex with the highest degree among rows and columns.
    Add  $v$  to the vertex cover
    Color  $v$  using star bicoloring algorithm
  end while
  Color all nodes that are not already colored with the color 0
end procedure

```

---

As can be seen the algorithm 4.6.1 does not bring major changes in the implementation itself but rather in the concept of the coloring. It has already been established the order in which the nodes are colored is essential, and that is exactly the element changed.

### Results

The results are presented in a Tabular form 4.2, along with a comparison with the basic star bicoloring algorithm and the SLO-ordered run of the star bicoloring algorithm. SLO was chosen as it has previously yielded the best results. The notations on the table are as follows:

- SBA - star bicoloring algorithm
- SBA+SLO - star bicoloring algorithm with smallest last ordering
- ISBA - integrated star bicoloring algorithm
- Delta SBA-ISBA Color difference between the algorithms. A positive number means the modified algorithm is better.

Matrix	#Colors SBA	#Colors SBA+SLO	#Colors ISBA	$\delta$ SBA-ISBA
barrier2-10	118	118	133	-15
bmwcra_1	186	138	177	9
c-73	57	57	57	0
cake12	96	68	72	24
cont-300	12	12	9	0
d_pretok	22	20	21	1
epb3	8	7	9	-1
ford2	38	33	36	2
hcircuit	23	21	14	9
lung2	8	8	14	-6
ohne2	204	204	167	37
para-5	162	162	128	34
pkustk13	303	300	207	96
rajat23	102	100	22	80
scircuit	86	85	43	43
ship_003	181	144	198	-17
shipsec1	126	114	150	-24
stomach	38	32	48	-10
torso2	18	13	21	-3
turon_m	23	20	22	1
twotone	105	89	90	15
xenon2	52	43	51	1
$\Sigma =$	1968	1785	1692	276

Table 4.2: Test results for integrated star bicoloring algorithm and the comparison to other significant algorithms

Whilst there are some matrices where the results are poorer, such as ship\_003 and shipsec1, this is more than made up for by the positive results with matrices such as pkustk13, rajat23 and scircuit. The total difference obtained is in favor of the new heuristic.

## Discussion

Upon discovering the promising results of the algorithm it was tried to discover why it behaved so much better on some of the matrices. After concluding the results were correct, the efforts were focused on the *rajat23* matrix, which showed the greatest difference, 22 compared to the previous 102 colors. Due to the size of the matrix information could not be gathered just by looking at the detailed output, but rather by trying to understand what makes this matrix different and try to reproduce it on a smaller scale. This means trying to find smaller matrices that have the same properties and continuing this process until the smallest matrix is found for which differences in results still appear.

*The rajat23 matrix* By looking at the Matrix 4.2 one can see the fact that it has quite a lot of elements on the last rows and columns. This means that a correct coloring should first take the last rows and columns and color them before moving on to the others. This was not the case for most implementations, as they always color either the rows and then the columns or vice versa, but never at the same time. This will of course have the effect that some nodes with a high degree will be colored not at the begin of the algorithm but rather (in the best case) somewhere on the way. Without any kind of preordering, they will quite probably be colored last, resulting in many colors added.

After coming up with a theory that explained the performance of the algorithm increasingly smaller parts of the *rajat23* matrix were used, and the hope existed that the phenomena will manifest itself on easier to understand matrices. The gambit was a success, as I was able to understand the differences in the ordering of the nodes that are colored which lead to different results. The theory is thus sound, and we are therefore left with a basic idea that works, but obviously needs improvement.

The idea central to this breakthrough is the following: the matrix has high-degree nodes both in rows and columns. If no preorderings are used the results are very poor as the high-degree nodes are among the last to be colored. If preorderings are used, the results improve, but not to the amount achieved by the integrated algorithm. The star bicoloring scheme colors all rows, then all columns. Whilst both rows and columns are colored using the preordering given, no columns will be colored at all until the rows are finished. Thus the reason the integrated algorithm is better is that it colors the high degree vertices of both rows and columns at once.

A reversal of the coloring order, that is to color all columns before moving on th

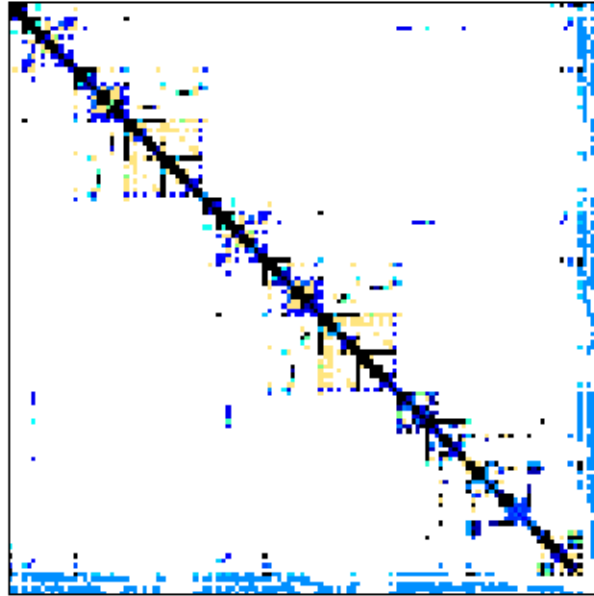


Figure 4.2: Rajat 23 Nonzero structure displayed using cspy

the rows, was tried at the possibility existed this could also bring something. It was doubtful any extremes would be good. Some testing, which is not included here because of the low value of the results shows that a column-row coloring is even worse than the row-column approach. The results were about 10 times worse.

The obvious conclusion is that neither extreme is effective, and that a mixed approach is better. This confirms the supposition made earlier.

## Conclusion

This new approach is promising and although the results are better than previous heuristics given the total results, on certain cases they are worse. I am hopeful that this heuristic can be improved upon, to alleviate the worst cases and perhaps even further improve on the best ones.

## 4.7 Total ordering star bicoloring algorithm

### Concept

After the initial discovery represented by the integrated star bicoloring scheme a venture was undertaken to eliminate the cases where it had poorer performance than the original. Now that it was realized that it was not the coloring while computing the independent set that was the cause of the success, but the ordering of the vertices, a return to the original program was made and the order that the nodes were chosen was simply changed. A simple degree based ordering is chosen, but the algorithm was modified to make that choice dynamically, and take into account both row and column nodes. This ordering is an improvement over the implementation in [Lül06] as [GMP05] gives no suggestions as to the ordering of the nodes.

### Algorithm

---

**Algorithm 4.7.1** Total ordering star bicoloring

---

**procedure** TOTALORDERINGSTARBICOLORING( $G_b = (V_1, V_2, E)$ )

    Determine Independent Set  $I$

    Color all nodes in  $I$  with the color 0.

    Let  $v$  be the vertex with the highest degree among rows **and** columns.

    Color  $v$  just as you would in the normal star bicoloring algorithm

**end procedure**

---

As can be seen the Algorithm 4.7.1 does not bring major changes in the implementation itself but rather in the concept of the coloring. It has already been established the order in which the nodes are colored is essential, and that is exactly what this program changes. In addition, the clean and elegant implementation has been preserved, and on this variation preordering may be used as well (though with limited effect)

### Results

The results are presented in the Table 4.3, along with a comparison with the basic star bicoloring algorithm and the modified run of the same. The notations on the table are as follows:

- SBA - star bicoloring algorithm

- ISBA - improved star bicoloring algorithm
- TOSBA - total ordering modified star bicoloring algorithm
- Delta TOSBA-ISBA Color difference between the algorithms. A positive number means the total ordering algorithm is better.

Matrix	#Colors SBA	#Colors ISBA	#Colors TOSBA	$\Delta$ TOSBA-ISBA
barrier2-10	118	133	119	14
bmwcra_1	186	177	165	12
c-73	57	57	65	-8
cage12	96	72	72	0
cont-300	12	9	12	0
d_pretok	22	21	21	0
epb3	8	9	9	0
ford2	38	36	34	2
hcircuit	23	14	16	-2
lung2	8	14	10	4
ohne2	204	167	155	12
para-5	162	128	120	8
pkustk13	303	207	236	-29
rajat23	102	22	29	-7
scircuit	86	43	47	-4
ship_003	181	198	156	42
shipsec1	126	150	126	24
stomach	38	48	35	13
torso2	18	21	21	0
turon_m	23	22	22	0
twotone	105	90	90	0
xenon2	52	51	51	0
$\Sigma =$	1968	1692	1613	88

Table 4.3: Test results for total ordering star bicoloring algorithm

## Discussion

It can be observed that the cases where the results were poorer have been substantially improved, while the cases where the results were better have largely remained the same. Not using any kind of  $\rho$  is sure to have an effect as well on the results, so a part of the skew can be explained by that.



## Conclusion

This new approach continues to be promising and I am certain that this heuristic can be improved upon, to take full advantage of the other types of preordering. This implementation only contains a mimicking of LFO, which has been proven to be the preordering with the poorest results. I am hopeful that SLO and IDO will bring better results.



## Chapter 5

---

# Restricted coloring for partial Jacobian computation

---

This chapter describes the algorithms for restricted graph coloring used as a starting point as well as those developed while working on the thesis and the patterns discovered while working on restricted Jacobian computation using bi-coloring algorithms. The source code is not included in this chapter but can be found in Appendix 2.

This chapter begins by presenting the algorithm used as a starting point as well as the one with the total ordering implemented, followed by a description of the patterns used to determine the required elements for restricted colorings of sparse Jacobian computations. The chapter ends with the results obtained using the patterns and algorithms presented.

### 5.1 Restricted star bicoloring algorithm

This is the algorithm, presented and implemented in [Lül06] is the starting point for my work in this area. It represents the state of the art in the field of restricted graph coloring for sparse Jacobian computation. I will introduce it here as a reference point to all my followin restriced algorithms.

Notice the structure of the Algorithm 5.1.1, which allows for easy modification while maintaining its intelligibility. The only difference when compared to

**Algorithm 5.1.1** Restricted Star Bicoloring Algorithm for  $G_b$ 


---

```

procedure RESTRICTEDSTARBICOLORING( $G_b = (V_r, V_c, E), Color$ )
  Let  $v_1, v_2, \dots, v_{|V_r \cup V_c|}$  a given ordering of  $V_r \cup V_c$ 
  Initialize Array forbiddenColors with  $-1$ 
  for  $i := 1, \dots, |V_r \cup V_c|$  do
    if  $Color[v_i] \neq 0$  then
      for each  $w \in N_1(v_i, G_b)$  do
        if  $Color[w] \leq 0$  then
          for each  $x \in N_1(w, G_b)$  with  $Color[x] > 0$  do
            if  $(v_i, w) \in E_R$  or  $(w, x) \in E_R$  then
              forbiddenColors[ $Color[x]$ ] :=  $i$ 
            end if
          end for
        else
          for each  $x \in N_1(w, G_b)$  with  $Color[x] > 0$  do
            if  $(w, x) \in E_R$  then
              for each  $y \in N_1(x, G_b)$  with  $Color[y] > 0$  and  $y \neq w$  do
                if  $Color[w] = Color[y]$  then
                  forbiddenColors[ $Color[x]$ ] :=  $i$ 
                end if
              end for
            end if
          end for
        end if
      end for
    end if
    Color[ $v_i$ ] :=  $\min\{j > 0: \text{forbiddenColors}[j] \neq i\}$ 
  end if
end for
  Add  $\max\{Color[v_r] : v_r \in V_r\}$  to all  $Color[v_c]$  with  $v_c \in V_c$  and  $Color[v_c] > 0$ 
  return Color
end procedure

```

---

the complete coloring scheme is that the only edges which cause a color to be discounted are those that are actually required in the final coloring. The algorithm also does not allow a nonzero element to be placed in the same partition as a required element if they share the same position, as that would lead to the required element being combined with the nonrequired nonzero element.

## 5.2 Restricted Total Ordering Star Bicoloring Algorithm

After proving that the total ordering star bicoloring algorithm is superior in the case of the complete coloring, the decision was made to move on to the restricted coloring and try to adapt the solution to that code. A simple degree based ordering was chosen, but the modified the algorithm is allowed to make that choice dynamically, and take into account both row and column nodes. The modifications to the restricted algorithm while subtle bring a great difference in the results.

---

**Algorithm 5.2.1** Restricted Star Bicoloring Algorithm for  $G_b$ 


---

```

procedure RESTRICTEDSTARBICOLORING( $G_b = (V_r, V_c, E), Color$ )
  Let  $v_1, v_2, \dots, v_{|V_r \cup V_c|}$  a given ordering of  $V_r \cup V_c$ 
  Initialize Array forbiddenColors with  $-1$ 
  while Not all nodes are colored do
    Let  $v_i$  be the vertex with the highest degree among rows and columns that is
    not colored.
    for each  $w \in N_1(v_i, G_b)$  do
      if  $Color[w] \leq 0$  then
        for each  $x \in N_1(w, G_b)$  with  $Color[x] > 0$  do
          if  $(v_i, w) \in E_R$  or  $(w, x) \in E_R$  then
             $forbiddenColors[Color[x]] := i$ 
          end if
        end for
      else
        for each  $x \in N_1(w, G_b)$  with  $Color[x] > 0$  do
          if  $(w, x) \in E_R$  then
            for each  $y \in N_1(x, G_b)$  with  $Color[y] > 0$  and  $y \neq w$  do
              if  $Color[w] = Color[y]$  then
                 $forbiddenColors[Color[x]] := i$ 
              end if
            end for
          end if
        end for
      end if
    end for
     $Color[v_i] := \min\{j > 0: forbiddenColors[j] \neq i\}$ 
  end while
  Add  $\max\{Color[v_r] : v_r \in V_r\}$  to all  $Color[v_c]$  with  $v_c \in V_c$  and  $Color[v_c] > 0$ 
  return Color
end procedure

```

---

The algorithm 5.2.1 does not bring major changes in the implementation itself but rather in the concept of the coloring. Much like in the complete coloring for full Jacobian computation, the same style of implementation has been preserved, and on this variation preordering may be used as well (though with limited effect). The only difference to the base algorithm is that both row and column elements can be colored, not just after all row elements have had colors assigned to them.

### 5.3 The main diagonal pattern

The first pattern that used in testing was the main diagonal pattern also known as the Jacobi preconditioner. This signifies that only the partial derivatives of the form  $\frac{\partial x}{\partial x}$  are significant to the problem and only these values must be determined.

In [Lül06] proof exists that using this kind of pattern the unidirectional coloring can always achieve the same results as bidirectional coloring. The Jacobi preconditioner, although widely used is as such not the nest candidate to try to test bidirectional coloring algorithms, and additional patterns had to be found.

### 5.4 The lattice pattern

The search for a relevant pattern that could show the need for a restricted bicoloring algorithm has lead to the sampling techniques used in various scientific computation problems. A simple sampling technique when the basis is a two dimensional matrix is the lattice. This implies choosing columns and rows, so that the elements of the initial matrix chosen create a mesh.

The first lattice implemented was a fine, 1 in 10 lattice, meaning that approximately 20% of the original Jacobian elements would be required. The number of elements is quite high, and as such it is expected to lead in some cases to a higher number of colors than a full coloring would bring. As an additional element, the main diagonal was also kept as a required element.

The second lattice structure was a course lattice, consisting of only 10 equidistant columns and rows and the main diagonal of the matrix.

Testing was done using the restricted D2 coloring algorithm for rows and columns and the restricted modified and unmodified star bicoloring algorithm.

## 5.5 The high-value element pattern

An interesting pattern that can appear in scientific computation optimization problems is that only the parameters that can have a major impact on the problem are actually desired. This translates into only elements of the Jacobian with high values need to be determined.

In practice the problems themselves are known enough as to know which input and output parameters are relevant and have to be determined. For this work, I have used MATLAB scripts to determine the larger elements of some of the test matrices that seemed suited to this approach and used these elements to form the pattern. In the following Figure [?] the pattern used for one of the matrices can be viewed alongside the original matrix.

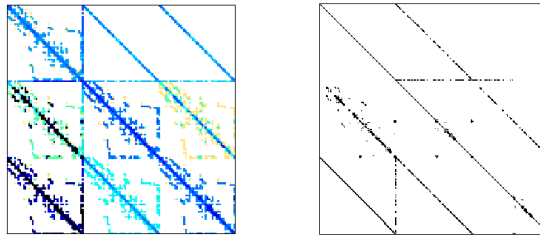


Figure 5.1: The ohne2 matrix is on the left and the elements required are on the right.

The results show that bidirectional coloring is superior to unidirectional coloring for a number of matrices. The improved algorithm also brings significant reduction in the number of colors used.

## 5.6 The diagonal block

The last pattern experimented with was the diagonal block pattern, useful for optimizing computations for running in parallel. This pattern is also quite frequent in the test matrices from the Tim Davies Collection.

Blocks of  $1000 \times 1000$  elements are chosen along the main diagonal of the chosen test matrices to be the required elements.

The results show that bidirectional coloring is superior to unidirectional coloring for a number of matrices. The improved algorithm also brings a noticeable reduction in the number of colors used.

## 5.7 Results

As a general observation pertaining restricted coloring, it is entirely possible to have worse results when trying to obtain many elements of the Jacobian in this way rather than complete cover. The method is created with the purpose of choosing few elements when compared to their total number in the matrix. There will be some such cases among the results, but as expected these are only limited exceptions which occur when the pattern is very dense in raport to the density of the matrix itself.

These are the results comparing the two star bicoloring algorithms, namely the starting one and the one with total ordering implemented, using the diagonal pattern and are presented in a Tabular form 5.1. The notations on the table are as follows:

- RSBA - restricted star bicoloring algorithm
- RTOSBA - restricted total ordering star bicoloring algorithm
- Delta RSBA-RTOSBA Color difference between the algorithms. A positive number means the modified algorithm is better.

The difference is not as great as for the case of the full JAcobian computation, for one because of the pattern used which does not allow for bidirectional coloring to really prove its value, and secondly, because the total number of color used is smaller. It is plain though, that using this heuristic yields different results in this case as well.

To really discover the worth of restricted bicoloring for sparse Jacobian computation the pattern used must be changed as the Identity pattern is inconclusive in this respect. The improvement of 2.8% is too small to allow for a real conclusion about the performance of the modified restricted coloring algorithm, and also the improvement in itself cannot prove or disprove the usefulness of bicoloring in restricted coloring problems. The attention will hereby be focused on trying to obtain relevant patterns, and the first among these is the lattice pattern.



Matrix	#Colors RSBA	#Colors RTOSBA	$\delta$ RSBA-RTOSBA
barrier2-10	26	28	-2
bmwcra_1	40	47	-7
c-73	2	2	0
cage12	14	13	1
cont-300	1	1	0
d_pretok	6	7	-1
epb3	5	5	0
ford2	28	27	1
hcircuit	5	5	0
lung2	4	5	-1
ohne2	36	35	1
para-5	27	28	-1
pkustk13	57	45	12
rajat23	9	4	5
scircuit	10	8	2
ship_003	60	54	6
shipsec1	48	54	-6
stomach	8	6	2
torso2	5	7	-2
turon_m	7	7	0
twotone	5	4	1
xenon2	20	19	1
$\Sigma =$	424	412	12

Table 5.1: Test results for restricted total ordering star bicoloring algorithm and compariosn with SBA

Test were run using lattice structures and as can be seen the effects are staggering. The number of colors needed increases linear in the case of bicoloring algorithms but a lot faster than linear in the case of unidirectional coloring. The tables 5.2, 5.3 show the great improvements represented by the bicoloring in these cases. The first table presents the results for the fine lattice, while the second the results for the course one. The notations on the table are as follows:

- RD2A - restricted distance 2 coloring algorithm
- RSBA - restricted star bicoloring algorithm
- RMSBA - restricted modified star bicoloring algorithm
- Delta RD2A-RMSBA Color difference between the algorithms. A positive number means the modified algorithm is better.

Matrix	#Colors RD2A	#Colors RSBA	#Colors RMSBA	Delta RD2A-RMSBA
barrier2-10	1743	1751	82	1661
c-73	29315	55	64	29251
cage12	62	65	58	4
cont-300	10	11	12	-2
epb3	8	8	8	0
ford2	34	33	31	2
hcircuit	1399	16	16	1383
ohne2	2428	1193	166	2262
para-5	1772	1783	126	1646
pkustk13	303	252	192	111
rajat23	208	35	19	189
scircuit	119	67	39	80
ship_003	167	98	120	47
shipsec1	126	89	89	37
torso2	14	15	15	-1
turon_m	10	25	24	-4
xenon2	48	45	46	2
$\Sigma =$	37776	5541	1107	36669

Table 5.2: Test results for restricted modified star bicoloring algorithm using fine lattice

The results are obviously in favor of the bicoloring algorithm and as such the results for a courser lattice were needed to ensure that the theorem emerging holds. The cases already on hand, the identity matrix and the fine lattice are extreme cases, one for a low number of elements required and the other for a large number of elements. This next test shows the middle ground, and proves that the theorem still holds.

The idea that becomes apparent through these results is that indeed, the number of colors used by restricted bicoloring increases linear with the percent of elements required, while the number of colors used by the restricted distance 2 coloring increases a lot faster. This makes restricted bicoloring a valid option in problems requiring only a part of the elements, if the pattern is more complex than the identity matrix.

The Table 5.4 shows improvements represented by the bicoloring in the case of the high-value pattern. Notice that the total ordering star bicoloring algorithm is an improvement over the star bicoloring algorithm. The notations on the table are as follows:

Matrix	#Colors RD2A	#Colors RSBA	#Colors RMSBA	$\delta$ RD2A-RMSBA
barrier2-10	63	29	30	33
c-73	13	9	12	1
cage12	26	15	14	12
cont-300	6	3	6	0
epb3	7	7	6	1
ford2	28	29	28	0
hcircuit	5	6	6	-1
ohne2	81	36	38	43
para-5	51	29	30	21
pkustk13	117	58	46	71
rajat23	8	7	7	1
scircuit	10	11	10	0
ship_003	90	61	55	35
shipsec1	55	49	55	0
torso2	10	8	8	2
turon_m	11	8	8	3
xenon2	27	22	20	7
$\Sigma =$	608	387	379	229

Table 5.3: Test results for restricted modified star bicoloring algorithm using course lattice

- RD2AR - restricted distance 2 coloring algorithm rows
- RD2AC - restricted distance 2 coloring algorithm columns
- RSBA - restricted star bicoloring algorithm
- RTOSBA - restricted total ordering star bicoloring algorithm

Matrix	#Colors RD2AR	#Colors RD2AC	#Colors RSBA	#Colors RMSBA
barrier2-10	1741	1422	1344	69
bmwcra_1	439	302	278	250
ohne2	2165	1622	843	143
para-5	2436	1772	997	126
rajat23	14	8	7	6
$\Sigma =$	6795	5126	3469	594

Table 5.4: Test results for total ordering modified star bicoloring algorithm using the high-value pattern

The Table 5.5 shows improvements represented by the bicoloring in the case of the diagonal block pattern. Notice that the total ordering star bicoloring algorithm is an improvement over the star bicoloring algorithm. The notations on the table are as follows:

- RD2AR - restricted distance 2 coloring algorithm rows
- RD2AC - restricted distance 2 coloring algorithm columns
- RSBA - restricted star bicoloring algorithm
- RTOSBA - restricted total ordering star bicoloring algorithm

Matrix	#Colors RD2AR	#Colors RD2AC	#Colors RSBA	#Colors RMSBA
barrier2-10	117	117	103	105
bmwcra_1	276	276	188	182
ohne2	476	476	145	138
para-5	101	101	111	117
rajat23	195	192	46	17
$\Sigma =$	1164	1161	593	559

Table 5.5: Test results for total ordering modified star bicoloring algorithm using the diagonal block pattern

## Chapter 6

---

### Conclusions

---

While creating this work I have focused on two areas of research, each with its own challenges and specific goals. The first area is full bidirectional Jacobian computation where the focus is trying to find improvements to the existing heuristics. The second area is restricted bidirectional Jacobian computation where the focus lies both on adapting the heuristic used in complete coloring to the restricted coloring and finding different patterns to test this coloring with.

The full bidirectional Jacobian computation has involved looking at the most advanced existing algorithms and trying to obtain improvements by combining the best concepts from the various heuristics with the star bicoloring scheme 4.1.1, considered to be the state of the art in the field.

By using different orderings than those already provided by the star bicoloring scheme an improvement has been obtained. For an overview, the best performance of the star bicoloring algorithm is a total of 1785 colors (that is, for all test matrices) and the new ordering 1613 colors were obtained. That means an improvement of 9,98% to the best heuristics up to date. For more detailed results consult Table 4.3.

The partial bidirectional Jacobian computation involved adapting the solutions obtained for complete graph coloring and testing to see the results. The effect was a small improvement over the already existing performance, mainly due to the nature of the pattern used in testing. By researching and using new patterns the effects were easier to observe. While for the basic pattern, represented by the main diagonal of the matrix in question, it has been proven that unidirec-

tional coloring is just as effective as bidirectional coloring, for the lattice, main diagonal block and high-value element pattern the results are quite different.

Using more complex patterns it quickly becomes apparent that the simple pattern was the reason for having unidirectional coloring as efficient as bidirectional coloring. The results are very promising showing large differences between unidirectional and bidirectional coloring and between the star bicoloring scheme and the total ordering variant.

The choice of pattern and ordering clearly affects the results obtained by the coloring algorithm, and here are the respective results, summarized in the following Table 6.1:

Pattern	Unidirectional	Bidirectional SBS	Bidirectional TO
Main diagonal	425	424	412
Lattice fine	37776	5541	1107
Lattice coarse	608	387	379
High-value elements	5126	3469	594

Table 6.1: Summarized test results for restricted graph coloring

I will also present some of the theoretical notions that have emerged during this work:

The previous findings have allowed me to come up with conjectures which may be used as a basis for continuing the improvement of the graph coloring heuristics. The first conjecture is in connection with the patterns observed and their density in relationship to the effectiveness of the partial distance 2 and partial bicoloring algorithms.

### Coloring efficiency and element density conjecture

*Conjecture* The percentage of the total elements required in the partial coloring of a Jacobian matrix is in direct proportion to the compared efficiency of the distance-2 and bicoloring algorithms. In other words, the greater the number of elements required in a restricted coloring, the better the bicoloring version performs in comparison to the one-colored version.

*Applications* Using this conjecture one could find out when to use the simpler partial distance-2 coloring scheme without reaching worse results. Since the bicoloring approach requires more time and resources and is also easier to implement knowing when this would suffice for a problem could be an important advantage.

### **3 colors on a distance 3 path conjecture**

*Conjecture* The number of colors used in a distance 3 path used in an optimal coloring scheme will always be 3.

*Theorem* A coloring scheme where the number of colors used in a distance 3 path is always 3 is not necessarily an optimal coloring scheme.

*Proof:* Any one sided coloring scheme has this property, thus it is obvious that that is not the optimal coloring scheme.

*Applications* Unfortunately this conjecture can not be so easily employed in a heuristic, as it is far more likely to end up with a one sided coloring than the optimal coloring scheme. This condition CAN be used though effectively in a branch-and-bound program as a stop condition, along with a restriction as to the number of colors used.





## Chapter 7

---

### Future work

---

Given the nature of the problem and the amount of research done so far in this field, it can safely be assumed that there is still a lot of work to be done. After working on this subject while writing my bachelors thesis there is still a lot that can be accomplished. There are improvements which could still be made to the algorithms without changing them altogether and this is only the beginning.

On the subject of full Jacobian computation using bipartite graph coloring the first improvement that could be made is to adapt the algorithm developed in this work to use the faster orderings, and even the dynamic orderings suggested in [HS98]. Alternatively, new heuristics could be devised that make better use of the particular structures of matrices, for example the block structure discussed earlier.

On the subject of partial Jacobian computation using bipartite graph coloring the patterns studied in this work are natural representations of some real life scientific modeling issues. Testing using other patterns could continue and by expanding the knowledge about them, more specialized effective heuristics could be developed.

In my opinion though, a great next step would be to integrate the framework obtained with an automatic differentiation tool to create an efficient and easy to employ tool for use in scientific computation. I believe it would be a great step forward in making graph-coloring based Jacobian computation better known.

I wish to continue researching this field in my future studies as I believe there is a lot that can still be accomplished. I hope to be able to further look into this theme during the continuation of my academic progress, especially into finding different heuristics and finding ways to determine beforehand what kind of heuristic would be better suited to what kind of Jacobian matrix.

---

# Bibliography

---

- [Adi] Adifor. Adifor website. <http://www.mcs.anl.gov/research/projects/adifor/>.  
[cited at p. 11]
- [AG08] A. Walther A. Griewank. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. SIAM, Philadelphia, PA, 2008.  
[cited at p. 9, 14]
- [Boo07] George Boole. *Calculus of finite differences*. Chelsea Publishing, 2007.  
[cited at p. 8]
- [CGM84] T. F. Coleman, B. S. Garbow, and J. J. More. Software for estimating sparse Jacobian matrices. *ACM Trans. Math. Softw.*, 10(3):329–345, 1984.  
[cited at p. 26]
- [CV98] T. F. Coleman and A. Verma. The Efficient Computation of Sparse Jacobian Matrices Using Automatic Differentiation. *SIAM J. Sci. Comput.*, 19(4):1210–1233, 1998. [cited at p. 16, 22]
- [GGMM00] Griewank, Andreas Griewank, Christo Mitev, and Christo Mitev. Detecting jacobian sparsity patterns by bayesian probing. Technical report, Math. Prog, 2000. [cited at p. 13]
- [GMP05] A. H. Gebremedhin, F. Manne, and A. Pothen. What Color Is Your Jacobian? Graph Coloring for Computing Derivatives. *SIAM Review*, 47(4):629–705, 2005. [cited at p. 17, 21, 34, 36, 37, 40]
- [HS98] S. Hossain and T. Steihaug. Computing a Sparse Jacobian Matrix by Rows and Columns. *Optimization Methods and Software*, 10:33–48, 1998.  
[cited at p. 26, 38, 40, 69]
- [Lül06] Michael Lülkesmann. Graphfärbung zur partiellen berechnung von jacobimatrizen. Master’s thesis, Fachgruppe Informatik, RWTH Aachen, 2006.  
[cited at p. 25, 26, 29, 36, 55, 58]

- [Ope] OpenAD. Openad website. <http://www.mcs.anl.gov/OpenAD/>.  
[cited at p. 11]
- [Ral81] L. B. Rall. *Automatic Differentiation: Techniques and Applications*, volume 120 of *Lecture Notes in Computer Science*. Springer, Berlin, 1981.  
[cited at p. 9]
- [Spi83] Murray R. Spiegel. *Advanced Mathematics for Engineers and Scientists*. Schaum, 1983. [cited at p. 8]
- [Tap] Tapenade. Tapenade website. <http://tapenade.inria.fr:8080/tapenade/index.jsp>.  
[cited at p. 11]

---

## List of Figures

---

1.1	Unidirectional coloring of a matrix and bipartite graph. For this coloring, complete unidirectional column coloring, the condition that must be upheld is that no two nodes that share a neighbor can have the same color. . . . .	4
2.1	(a) Computational graph for $f(x_1, x_2) = x_1 * x_2 + \cos(x_1)$ ; (b) Computation steps for $f(x_1, x_2) = x_1 * x_2 + \cos(x_1)$ . . . . .	11
2.2	(a) Computation graph for $\dot{f}(x_1, x_2) = \dot{x}_1 * x_2 + x_1 * \dot{x}_2 - \sin(x_1)$ ; (b) Computation steps for $\dot{f}(x_1, x_2) = \dot{x}_1 * x_2 + x_1 * \dot{x}_2 - \sin(x_1)$ . . . . .	12
2.3	(a) Computation graph for $f(x_1, x_2) = x_1 * x_2 + \cos(x_1)$ with adjoint objects ; (b) Computation steps for $f(x_1, x_2) = x_1 * x_2 + \cos(x_1)$ with adjoint objects . . . . .	13
2.4	The vectors are chosen in such a way that $J$ has no nonzero elements on the same row in this column-wise compression. $S$ is composed of the new partial derivation vectors. The resulting compressed matrix is $\tilde{J}$ . . . . .	15
2.5	$B$ : Column partition of a matrix $B$ using 3 partitions. The seed matrix $S$ has been used to determine the partitions of $B$ . The partitions are 1, 2, 3, 4 and 5 . . . . .	16
2.6	$B$ : Row partition of a matrix $B$ using 2 partitions. The seed matrix $S$ has been used to determine the partitions of $B$ . The partitions are 1, 2 and 3, 4, 5 . . . . .	16
2.7	$B_u$ : Unidirectional partitioning of $B$ . Notice that 5 colors are needed. $B_b$ : Bidirectional partitioning of $B$ . Notice that only 4 colors are needed. The partitions are 1 for rows and 1, 2, 3 and 4, 5 for columns	17

2.8	$G_1$ , an uncolored graph and $G_2$ a colored graph, using a distance-1 coloring. In this example the top node was colored first followed by the next row and finally the bottom row. Other coloring schemes are possible, optimal and otherwise. . . . .	19
2.9	Representation of a matrix $M$ as a bipartite graph $G$ by creating nodes for rows and columns and then adding edges between rows and column nodes for each nonzero element in the matrix. . . . .	20
2.10	Bipartite graph and unidirectional coloring of a matrix. The condition that must be upheld is that no two nodes that share a neighbor can have the same color. . . . .	21
2.11	Bipartite graph resulting of a matrix. The vertex cover is represented with horizontal hashing while the independent set is represented with a vertical hashing. . . . .	23
2.12	Examples of matrices from the Tim Davies Collection . . . . .	25
2.13	Bipartite graph $G$ and restricted unidirectional coloring of a matrix $M$ . The blacked out elements in the matrix are those which are of no interest. Notice that a fewer number of colors are needed in comparison to the example in Figure 2.10 . . . . .	29
3.1	Cluster information from the RZ homepage . . . . .	33
4.1	Example showing how a color can be eliminated. Note that this is just a small extract of a larger graph, only to present the concept. . .	46
4.2	Rajat 23 Nonzero structure displayed using cspy . . . . .	50
5.1	The ohne2 matrix is on the left and the elements requiered are on the right. . . . .	59

---

## List of Tables

---

2.1	Partial derivative operations $+$ , $*$ and $\cos()$ . . . . .	11
2.2	24 Matrices from the Tim Davis Matrix Collection with their major characteristics: number of columns, number of nonzeros, symmetry and sparsity . . . . .	24
2.3	Number of colors obtained with the <i>StarBicoloringSchema</i> without preordering and with preordering <i>LFO</i> , <i>SLO</i> and <i>IDO</i> . . . . .	27
2.4	Sum of colors obtained by the various algorithms . . . . .	29
2.5	Number of colors (without the color 0) obtained by uni- and bidirectional restricted computation algorithms . . . . .	30
4.1	Test results for complete direct cover algorithm, star bicoloring scheme and the comparison of the results . . . . .	40
4.2	Test results for integrated star bicoloring algorithm and the comparison to other significant algorithms . . . . .	48
4.3	Test results for total ordering star bicoloring algorithm . . . . .	52
5.1	Test results for restricted total ordering star bicoloring algorithm and compariosn with SBA . . . . .	61
5.2	Test results for restricted modified star bicoloring algorithm using fine lattice . . . . .	62
5.3	Test results for restricted modified star bicoloring algorithm using course lattice . . . . .	63
5.4	Test results for total ordering modified star bicoloring algorithm using the high-value pattern . . . . .	63
5.5	Test results for total ordering modified star bicoloring algorithm using the diagonal block pattern . . . . .	64
6.1	Summarized test results for restricted graph coloring . . . . .	66

---

## Listings

---