THE UNIVERSITY OF AUCKLAND
LIBRARIES AND LEARNING SERVICES

http://researchspace.auckland.ac.nz

ResearchSpace@Auckland

Copyright Statement

The digital copy of this thesis is protected by the Copyright Act 1994 (New Zealand).

This thesis may be consulted by you, provided you comply with the provisions of the Act and the following conditions of use:

- Any use you make of these documents or images must be for research or private study purposes only, and you may not make them available to any other person.
- Authors control the copyright of their thesis. You will recognise the author's right to be identified as the author of this thesis, and due acknowledgement will be made to the author where appropriate.
- You will obtain the author's permission before publishing any material from their thesis.

To request permissions please use the Feedback form on our webpage.
http://researchspace.auckland.ac.nz/feedback

General copyright and disclaimer

In addition to the above conditions, authors give their consent for the digital copy of their work to be used subject to the conditions specified on the Library Thesis Consent Form and Deposit Licence.

Department of Computer Science
The University of Auckland
New Zealand

THE UNIVERSITY OF AUCKLAND
NEW ZEALAND
Te Whare Wānanga o Tāmaki Makaurau

Constraint Solving for User Interface Layout Problems

Noreen Jamil

February 2015

Supervisors:

Christof Lutteroth

Gerald Weber

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS OF DOCTOR OF PHILOSOPHY IN SCIENCE

Abstract

This thesis proposes efficient and robust algorithms for solving linear constraints problems for graphical user interface (GUI) layout. Constraints have been recognised as a powerful method for the specification of GUI layouts for a long time. Most constraint problems encountered in this area are sparse, i.e. most linear coefficients in the corresponding coefficient matrix are zero. The numerical methods that have been proposed for solving layout problems so far are mainly direct methods. The focus of this research is on investigating iterative methods for solving these problems efficiently. The algorithms developed in this thesis are compared to well-known direct and linear programming algorithms. The performance and convergence of the proposed and existing algorithms were evaluated empirically using randomly generated UI layout specifications of various sizes.

Successive Over-Relaxation (SOR) is one of the most commonly used iterative methods for solving linear problems, and this was the first algorithm investigated in this thesis. In contrast to direct methods such as Gauss-elimination or QR-factorization, SOR is particularly efficient for problems with sparse matrices, as they appear in GUI layout specifications. However, SOR as described in the literature has its limitations: it works only with square matrices and does not support soft constraints, which makes it inapplicable to UI layout problems. This research extends SOR so that it can be used to solve non-square matrices and soft constraints.

Furthermore, different optimizations of SOR were investigated to speed up its convergence. First, we propose a metric to measure the optimality of a constraint sequence and then a Simulated Annealing based algorithm, that optimizes the order in which constraints are solved. Second, we propose Constraint-Wise Under-relaxation (CWU), a technique in which the relaxation parameters of individual constraints are adjusted during solving. Third, we investigate the use of a warm start strategy, which reuses the solution of a previous layout to warm start the solving of a new layout, taking into account that most layout changes during runtime are small.

Another contribution of this thesis is an investigation of the Kaczmarz method – an iterative orthogonal projection algorithm – for solving GUI layout problems. This algorithm is more efficient than the SOR algorithm and its convergence is guaranteed. However, to make it applicable to GUI problems some extensions were necessary, such as support for soft constraints. We also investigate some approaches for least-squares solving and optimization in this context.

Acknowledgements

There are many people who helped me during my studies, and I would like to take this opportunity to thank them.

Firstly, I would like to thank my supervisors, Dr. Christof Lutteroth and Dr. Gerald Weber, for their supervision and support during my PhD studies. Their professional help from the preliminary to the concluding stages enabled me to enhance my subject knowledge and polish my research skills. They have been very helpful throughout my studies, and especially in my first year. They always made time for me to talk about ideas and review papers.

I would like to thank administrators from my department namely, Heather Armstrong, Sithra Kumar and Robyn Young for their administrative support during my study. I would also like to thank Clemens Zeidler and Sudheendra Pulla for their unwavering support during my studies.

I was fortunate enough to collaborate with two researchers: Johannes Muller and Deanna Needell. I am extremely grateful to Johannes Muller for his help while he was a postdoctoral researcher at the University of Auckland. I have learned a lot from this experience and gained a deeper understanding of my area of research. The insightful comments of Deanna Needell of the Mathematics and Computer Science Department, Claremont McKenna College on my research publications were very helpful to me.

I am very thankful to Dr. Allison Heard from the Mathematics Department for proof-reading some of my thesis. My friends have helped to keep me sane throughout my studies. Our time spent together has been a blessing, and I thank them for everything they have done for me.

As I consider my doctoral thesis the most significant achievement in my life, I have no words to thank my parents for their love and support. My thanks, sincere prayers, and a lot of love and gratitude to my parents for their efforts to educate me. My daughter, Sheza, has helped me to keep joy in my life. She cooperated with me a lot throughout my studies. My husband, Asif, was very supportive during my studies. Many special thanks to him for managing all the activities that would have hindered my progress. I am sure we will remember this period of our lives for a long time. We all spent a cherishable time together in Auckland.

I am extremely thankful to Allah Almighty for his blessings, without his permission all this would not have been possible for me.

I dedicate this thesis to Sheza Naeem and Muhammad Asif Naeem.

Contents

1 Introduction **1**

1.1 Constraint-based GUI Layout . . . 2

1.2 Research Questions . . . 4

1.3 Structure of the Thesis . . . 6

2 Background: Solution Techniques for Linear Constraints **7**

2.1 Direct Methods . . . 8

2.1.1 LU-decomposition Method . . . 8

2.1.2 QR-decomposition Method . . . 9

2.2 Indirect Methods . . . 10

2.2.1 Jacobi Method . . . 10

2.2.2 Gauss-Seidel Method/ Successive Over-Relaxation . . . 11

2.2.3 Termination Criteria . . . 13

2.2.4 Advantages . . . 14

2.2.5 Convergence . . . 14

2.3 Linear Programming . . . 17

2.3.1 Simplex Method . . . 18

2.4 Summary . . . 19

3 Extending Successive Over-Relaxation for Non-Square Matrices **21**

3.1 Introduction . . . 21

3.2 Inequalities . . . 22

3.3 Non-Square Matrices . . . 23

3.4 Related Work . . . 23

3.5 Pivot Assignment . . . 24

3.5.1 Pivot Assignment Algorithms . . . 25

3.6 Summary . . . 27

4 Extending Successive Over-Relaxation for Soft Constraints **29**

4.1 Introduction . . . 29

4.2 Related Work 31
4.3 Prioritized IIS Detection 33
4.4 Prioritized Deletion Filtering 33
4.5 Prioritized Grouping Constraints 35
4.6 Experimental Evaluation 38
4.6.1 Methodology 38
4.6.2 Results 40
4.6.3 Discussion 44
4.7 Summary 44

5 Constraints Reordering Technique for Successive Over-Relaxation 47
5.1 Introduction 47
5.2 Related Work 48
5.3 Background: Effects of Constraint Reordering on Convergence of SOR 49
5.4 Optimizing the Solving Sequence of Constraints 50
5.4.1 Metric *goodness of solving sequence* (g) 52
5.4.2 Algorithm Based on Simulated Annealing 53
5.5 Experimental Evaluation 54
5.5.1 Methodology 54
5.5.2 Results 55
5.5.3 Discussion 57
5.6 Summary 57

6 Constraint-wise Under-relaxation 59
6.1 Introduction 59
6.2 Related Work 60
6.3 Experimental Evaluation 61
6.3.1 Methodology 61
6.3.2 Results 61
6.3.3 Discussion 64
6.4 Summary 64

7 Kaczmarz Algorithm with Soft Constraints 65
7.1 Introduction 65
7.2 Related Work 67
7.3 Kaczmarz Method 68
7.3.1 Convergence 68
7.3.2 Inequalities 70
7.4 Optimization of Kaczmarz 71

7.5 Least Squares Kaczmarz for Constraints Solving for Attractive GUIs 71
7.5.1 Least Squares Kaczmarz Method using a Cooling Function 73
7.5.2 Distinguishing Hard and Soft Constraints 74
7.5.3 Solution Technique for Hard and Soft Constraints 75
7.6 Experimental Evaluation 77
7.6.1 Methodology 77
7.6.2 Results 78
7.6.3 Discussion 80
7.7 Summary 81

8 Speeding Up Kaczmarz and Successive Over-RelaxatioS with a Warm-Start Strategy 83
8.1 Introduction 83
8.2 Related Work 84
8.3 Warm-Start Strategy 84
8.4 Experiment 86
8.4.1 Methodology 86
8.4.2 Results 87
8.5 Summary 89

9 Framework Design 91
9.1 Auckland Layout Model (ALM) 91
9.1.1 `ALMLayout` 92
9.1.2 `Variable, XTab and YTab` 92
9.1.3 `Row and Column` 92
9.2 linsolve 94
9.2.1 `LinearSpec` 94
9.2.2 `LayoutSpec` 94
9.2.3 `Constraint` 94
9.2.4 `LinearSolver` 94
9.2.5 `AbstractLinearSolver` 94
9.2.6 `AbstractSoftSolver` 95
9.2.7 `SOR` and `PivotSummandSelector` 95
9.2.8 `AbstractMatlabSolver` 95

10 Conclusion and Future Work 97
10.1 Achievements 97
10.2 Future Directions 99
10.3 Applications of Iterative Methods 100

10.3.1 Saddle Point Problems 100
10.3.2 Computerized Tomography 100
10.4 Reflections 101

List of Figures

1.1 Example constraint-based UI layout with hard and soft constraints 3

2.1 Sparse matrix example 10
2.2 Convergence and divergence behaviour of Gauss-Seidel 13

4.1 Example of constraint-based UI layout with hard and soft constraints 30
4.2 Example run of the prioritized grouping constraints algorithm 37
4.3 Performance comparison of prioritized IIS detection, prioritized deletion filtering and prioritized grouping constraints using random pivot assignment 41
4.4 Runtime comparison of prioritized IIS detection, prioritized deletion filtering and prioritized grouping constraints with deterministic pivot assignment 42
4.5 Performance comparison of the best solving strategies with LINPROG, LP-Solve and QR-decomposition 43

5.1 Graphical representation of problem with 4 constraints 52
5.2 Simulated Annealing algorithm to minimize g with $random()$ randomly selecting elements and $N()$ defining the neighborhood of a given solution (adapted from [100]) 53
5.3 Runtime comparison of LINPROG, LP-Solve and QR-decomposition with our new solvers 55
5.4 Runtime with and without sequence optimization 56

6.1 Performance comparison of the best solving strategies with LINPROG, LP-Solve and QR-decomposition 62
6.2 Performance comparison of SOR with and without CWU 63

7.1 Kaczmarz method overview 69
7.2 Kaczmarz method convergence 69
7.3 Two different solving strategies for a simple three-button layout 72
7.4 Example: Least Squares Kaczmarz method using a cooling function 73

7.5 Performance comparison of Kaczmarz with prioritized IIS detection, Kaczmarz with prioritized grouping constraints, Kaczmarz with CWU, Least Squares Kaczmarz with cooling function and SOR with random pivot assignment 79
7.6 Performance comparison of the best solving strategies with LINPROG, LP-Solve, QR-decomposition and prioritized IIS detection using random pivot assignment 82

8.1 A GUI constraint specification before and after resizing 85
8.2 Small-step resizing performance results 88
8.3 Big-step resizing performance results 89
8.4 Constraint change performance results 90

9.1 Class Diagram of different components involved in alm and linsolve. 93

List of Tables

4.1 Symbols used for the performance regression model 40
4.2 Regression models for the different solving strategies 41
4.3 Comparison of the quality of the solutions produced by the different algorithms (rank 1 = best) 43

5.1 Symbols used for the performance regression model 56
5.2 Regression models for the different solving strategies 57

6.1 Symbols used for the performance regression model 62
6.2 Regression models for the different solving strategies 63

7.1 Symbols used for the performance regression model 80
7.2 Regression models for the different solving strategies 80
7.3 Comparison of the quality of the solutions produced by the different algorithms (rank 1 = best) 80

8.1 Symbols used for the performance regression model 87
8.2 Regression models for solvers with and without warm-start strategy 88

1 Introduction

Graphical user interfaces (GUIs) are one of the most important modes to interact with a computer. To use the screen real estate efficiently, developers need to specify the positions and sizes of a GUIs widgets. Constraint-based approaches have long been seen as a desirable method in the construction of GUIs, where they are mainly used to define the layout of a GUI [86]. Several constraint-based GUI layout technologies have been developed and each of these technologies has their own peculiarities and requires specific knowledge. The constraint idea was introduced by Sutherland in 1960's Sketchpad [116], the first interactive graphical interface that solved geometric constraints. Since then many constraint solvers have been developed and studied by the research community [11,22,65,86,127], and interest has increased with the recently introduced constraint-based layout model in the Cocoa API of Apple's Mac OS X[1].

One of the challenges of the last few decades has been the construction of fast numerical algorithms for solving GUI layout problems. Recent developments in iterative methods have improved their efficiency. As a consequence, the use of iterative methods has become ubiquitous in recent years for solving sparse, real-world optimization problems where direct algorithms are not suitable due to fill-in, i.e. coefficients change from an initial zero to a non-zero value during the execution of the algorithm [16]. Unlike direct algorithms, which try to solve the problems in a finite number of steps, iterative methods start with a complete but preliminary approximation that is not necessarily consistent. They improve this approximation in several iterative steps until specified stopping criteria are satisfied. We can get good approximate solutions with iterative

[1] Cocoa Auto Layout Guide, 2012: `http://developer.apple.com`

methods, which is useful for practical applications especially if an efficient solution is required.

The linear problems arising in the domain of constraint-based layout are known to be sparse. For sparse linear problems, iterative methods are known to perform very well. All current approaches use direct methods and linear programming techniques for solving GUI layout problems. The most used popular linear programming technique, the Simplex algorithm, typically uses an iterative method along with one step of a direct method for each iteration. Direct methods suffer from fill-in effects when solving sparse systems, which generally makes them inferior to iterative methods in these cases. This means when using a direct method with more complex GUI specifications the responsiveness of the GUI decreases due to increasing computational effort. To decrease the computational effort this research investigates the application of iterative methods. Iterative methods are further classified into stationary, orthogonal projection and Krylov subspace methods. Krylov subspace methods such as conjugate gradient are not immediately applicable for solving the problems arising in GUI layout. Instead, this research limits its focus to stationary iterative and orthogonal projection methods because these methods are often simpler, faster and easy to implement than other methods for solving large sparse linear constraint problems. Numerical results presented in this thesis are based on problems arising from GUI layout applications, but the methods used in this thesis can also be used in other application areas.

1.1 Constraint-based GUI Layout

GUIs are commonly created with the help of GUI toolkits, i.e. libraries which implement the most common widgets and functions that are required in GUIs. In early GUI toolkits (e.g. Responsive web design (RWD)), developers needed to set the position and size of each widget manually, and also compose code that manages these values during the runtime of an application. For example, if the size of a window changes, the code would typically reposition and resize the widgets in that window. This static approach can become very tedious during the design process where GUI widgets are moved around frequently. In this case already placed widgets have to rearranged when inserting a new widget.

Many modern GUI toolkits incorporate layout managers, which support developers in the specification of resizable layouts. Rather than dealing with the position and size of each widgets manually, developers can feed a layout manager with a more abstract layout specification, and the layout manager then repositions and resizes the widgets automatically. It becomes easier for developers to modify a GUI because individual widgets do not have to be rearranged manually. GUIs can also become more consistent with the use of a layout manager; its use can help to ensure proper alignment and consistent spacing between adjacent widgets.

A GUI toolkit in the present age often offers different layout managers, of which some support very simplistic layouts while others support more sophisticated ones. A flow layout

would, for example, simply take a list of controls and arrange them on a panel in rows, beginning a new row when the current one has insufficient space. The Gridbag layout, which is a more complex layout manager, arranges all controls in a table, possibly with additional constraints. Constraint-based layout models specify layouts mathematically as constraint problems. They are among the most powerful layout models. The Auckland Layout Model (ALM) [87] is an example of a constraint-based layout models and is used in this thesis without loss of generality.

Constraints involve linear systems of equations and inequalities. Such system of linear equations is a set of m linear equations and inequalities in n variables. Generally, such systems can be written as

$Ax = b$,

$Ax \geq b$,

where A is a coefficient matrix, x is a variable and b is a right hand side vector. If $m < n$ then the system is called under-determined and there is no solution or infinitely many solutions. If $m = n$ then the system may have a unique solution and sometimes no solution. If $m > n$ then the system is called over-determined and may have no solution.

Constraints are a suitable mechanism for specifying the relationships among objects. They are used in the area of logic programming, planning and other optimization problems. They can be used to describe problems that are difficult to solve, conveniently decoupling the description of the problems from their solution. Due to this property, constraints are a promising way of specifying UI layouts, where the objects are widgets and the relationships between them are spatial relationships such as alignment and proportion. In addition to its relationships to other widgets, each widget has its own set of constraints describing properties such as minimum, maximum and preferred size. GUI layouts are often specified with linear constraints [86]. The positions and sizes of the widgets in a layout translate to variables. Constraints about alignment and proportions translate to linear equations, and constraints about minimum and maximum sizes translate to linear inequalities.

In ALM, there are constraints for each widget that relate each of its four boundaries to another part of the layout, or specify boundary values for the widget's size, as shown in Figure 1.1.

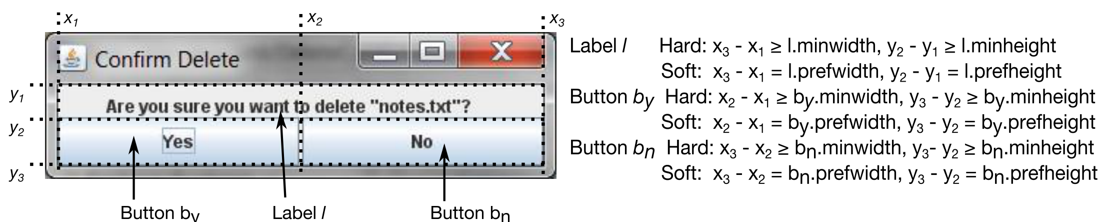


Figure 1.1: Example constraint-based UI layout with hard and soft constraints

A complete layout in ALM is defined by a set of areas (defined by a set of tabstops) and a set of constraints. With a given set of constraints ALM layout manager has to solve a system

of linear equalities and inequalities. It is often the case in layout specifications that the constraint system is over-specified. To cope with that problem ALM introduces the notion of soft constraints [86].

In contrast to *hard* constraints, which cannot be violated, soft constraints may be violated as much as necessary if no other solution can be found. To solve layouts which are defined with soft constraints it is not sufficient to solve a system of linear equations and a type of optimization, namely minimization of the constraint-violation, must be introduced. The violation is modelled with an introduced penalty parameter for each soft constraint.

The Auckland Layout Manager has a solid mathematical foundation. GUI developers can specify a layout on different layers of abstraction. On the lowest level of abstraction, layouts are specied by linear constraints and a linear objective function. Optimal layouts are calculated using linear programming. This means that all layouts are ultimately specified in terms of linear algebra.

The function of the next layer of abstraction is to manage rectangular areas of the GUI, which may contain controls or other graphical elements. In this layer, ordinal constraints are used to specify the topology of the elements in the GUI, and domain-specific parameters are used for size constraints of an area. The layer of areas is on the top of layer of soft constraints.

On the top level, a layout specification offers abstractions for rows and columns in which areas can be aligned, similar to common spreadsheet applications. This layer employs domain-specific parameters for the alignment of areas in table cells.

1.2 Research Questions

The main aim of this thesis is to provide an efficient algorithm for solving linear constraints for GUI layouts. We give an overview of existing algorithms for solving linear constraints. The detailed review of these algorithms has motivated an investigation of well-known iterative algorithms: Successive Over-Relaxation (SOR) and Kaczmarz. In this thesis we investigate the following questions and provide possible answers for them.

Q1: How can non-square GUI layout specifications be solved with SOR? Most specifications in GUI layout are non-square. For example, when specifying GUI layout with linear constraints, there are generally more constraints than variables. One of the most common iterative methods used to solve sparse linear systems is SOR. However, SOR in its original form does not work for solving non-square systems. To solve these systems we investigate three pivot assignment algorithms that can be used with any problem matrix, regardless of its shape or diagonal elements [71]. The first algorithm chooses pivot elements pseudo-randomly. In the second algorithm pivot elements are selected deterministically by optimizing certain selection criteria. The third algorithm chooses the “best” constraint for a variable and the “best” variable

for a constraint according to some formal criteria. We evaluate convergence and performance of these algorithms experimentally using randomly generated GUI layout specifications.

Q2: How can inconsistent GUI layout specifications be resolved? Most GUI layout specifications are inconsistent (e.g. the preferred size constraints). To resolve inconsistent problems, *soft constraints* are introduced. In contrast to the usual *hard* constraints, which cannot be violated, soft constraints may be violated if no other solution can be found. We investigate three algorithms for solving soft constraints with the SOR method [71]. The first algorithm successively adds non-conflicting constraints in descending order of priority. The second algorithm starts with all constraints and successively removes conflicting constraints in ascending order of priority. The third algorithm is a mixture of both and adapts the binary search algorithm to the problem of searching the maximum number of non-conflicting constraints. we present an experimental evaluation of these algorithms for randomly generated GUI layout specifications.

Q3: How can the sequence of constraints be reordered to speed up the convergence of SOR? The order of the constraints can have an effect on the speed of convergence of the SOR method. Determining the optimal sequence of solving constraints could help speed up the convergence. We propose an algorithm to optimise the sequence in which the constraints are solved [73]. Our contribution consist of, first, a metric to measure the optimality of a constraint sequence and, second, a simulated annealing based algorithm that optimizes the order of constraints. As mentioned earlier, we use usual empirical evaluation methods for randomly generated GUI layout specifications.

Q4: How can the convergence of iterative methods be improved, i.e. avoiding divergence? SOR has a relaxation parameter that can be helpful in speeding up its convergence. We investigate whether the convergence of SOR can be improved if we choose different relaxation parameters for each constraint. We show empirically that we can increase the computational performance of an SOR-based constraint solver if we choose different relaxation parameters for each constraint.

Q5: Can the Kaczmarz algorithm be applied for solving constraints in GUI layout problems? We present a variant of the Kaczmarz method for solving non-square matrices that can be applied to GUI layout problems. In its original form the Kaczmarz algorithm cannot handle soft constraints. Therefore we introduce several algorithms to solve soft constraints with Kaczmarz [74]. If we use Kaczmarz during resizing of a window in a GUI then the system can also be under-determined. In this case, space is not distributed in an asethetically pleasing way. To distribute the space according to the preferred size of the layout, we introduce the least squares Kaczmarz method to get the desired results. They are experimentally evaluated with regard to

convergence and performance, using randomly generated UI layout specifications.

Q6: When solving GUI layout problems what are the effects of warm starts on Kaczmarz and SOR based constraint solvers? GUI specifications change very little while programming is used during resizing of a window. The constraint solver has to calculate a new layout every time a GUI is resized, so it needs to be efficient to ensure a good user experience. We use Kaczmarz and SOR for solving GUI layout problems. We propose a warm start strategy to increase the computational performance of Kaczmarz and SOR based constraint solvers. In this strategy, we reuse the solution of a previous layout to warm start the solving of a new layout [72]. We measure experimentally the solving time of warm start strategy for randomly generated GUI layout specifications of various sizes.

1.3 Structure of the Thesis

The thesis is organized as follows. In Chapter 2 we give a detailed overview of the different numerical methods for solving linear constraint problems. These numerical methods are classified into several categories, each of which is described in detail. In Chapter 3, we extend SOR for solving non-square matrices for GUI layout. In Chapter 4, we propose three conflict resolution algorithms for solving systems of prioritized linear constraints with the SOR method. Then in Chapter 5, we propose an algorithm to optimize the sequence of solving constraints to speed up the convergence of the SOR method for conflict free non-square systems. This chapter presents first, a metric to measure the optimality of a constraint sequence and, second, a Simulated Annealing based algorithm, which optimizes the order of constraints. In the Chapter 6, we show empirically that if we use Constraint-Wise Under-relaxation, then convergence of SOR can be improved. We extend Kaczmarz for solving soft constraints for GUIs in Chapter 7. In Chapter 8, we give a description of the warm start strategy that can be used to improve the convergence behaviour of Kaczmarz and SOR-based constraint solvers for GUIs. In Chapter 9 we give a brief overview of implementation details of ALM and algorithms used for solving linear constraint problems. Chapter 10 concludes this thesis by summarising the achievements and providing some future guidelines.

2

Background: Solution Techniques for Linear Constraints

In this chapter we present a brief overview of the various classes of solution techniques that can be used to solve linear constraints. Linear problems are encountered in a variety of fields such as engineering, mathematics and computer science and hence various numerical methods have been introduced to solve them. These methods can be divided into direct and indirect, also known as iterative, methods. Direct methods aim to calculate an exact solution in a finite number of operations, whereas iterative methods begin with an initial approximation and usually produce improved approximations in a theoretically infinite sequence whose limit is the exact solution [109].

Many linear problems are sparse, i.e. most of the linear coefficients in the corresponding coefficient matrix are zero so that the number of non-zero coefficients is $O(n)$, with n being the number of variables [80] (see Fig 2.1). Since it is useful to have efficient solving methods specifically for sparse linear systems, much attention has been paid to iterative methods, which are preferable for such cases [5]. Iterative methods do not spend processing time on coefficients that are zero. Direct methods on the other hand, usually lead to fill-in, i.e. coefficients change from an initial zero to a non-zero value during the execution of the algorithm. In these methods the sparsity property is therefore lost and a lot more coefficients are involved, which makes processing slower. Although there are some techniques to minimize fill-in effects, iterative, indirect methods are often faster than direct methods for large sparse problems [16].

Linear constraints encountered in GUI layout are usually large and sparse, and numerous numerical methods have been introduced to solve them. There is no single method that is best for all situations. Methods should be appraised according to their speed and accuracy. Speed is an important factor in solving large systems of equations because the volume of computations involved is huge. Another issue is the accuracy of the round off errors involved in executing these computations. The following sections give an overview of solution methods that can be used to solve linear constraints, in order to set the stage and introduce fundamental terminology used in this thesis.

2.1 Direct Methods

Direct methods aim to calculate an exact solution in a finite number of operations. They are suitable for dense matrices, i.e. where the number of coefficients is $O(n^2)$ where n is the number of variables. The two best and widely used direct methods are described below:

- LU-decomposition method
- QR-decomposition method

2.1.1 LU-decomposition Method

LU-decomposition [28] is based on the fact that a non-singular square matrix A can be written as the product of lower triangular and upper triangular matrices. This method is also known as the LU-factorization method.
LU-decomposition is actually a variant of the Gaussian elimination method [28]. Consider the following linear systems of equations.

$$\begin{array}{ccc} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 & \cdots & a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 & \cdots & a_{2n}x_n = b_2 \\ \vdots & \vdots & \vdots \\ a_{n1}x_1 + a_{n2}x_2 + a_{n3}x_3 & \cdots & a_{nn}x_n = b_n \end{array}$$

which can be written as follows

$$Ax = b \tag{2.1}$$

Then *A* takes the form,

$$A = LU, \tag{2.2}$$

where *L* is a lower triangular matrix and *U* is an upper triangular matrix. So equation (2.1) becomes

$$LUx = b \tag{2.3}$$

and we can write

$$Ux = y \tag{2.4}$$

Thus the equation (2.3) becomes

$$Ly = b \tag{2.5}$$

and it proceeds as follows: first solve y in (2.5) by using the forward stage and then solve x in (2.4) by using the backward stage. The forward and backward stages are described as follows

1. Forward Stage

2. Backward Stage

1. Forward Stage: This involves the manipulation of equations in order to eliminate some unknowns from the equations and constitute an upper triangular system or echelon form.
2. Backward Stage: This stage uses the back substitution process on the reduced upper triangular system and produces the actual solution of the equation.

LU-decomposition is numerically unstable but its instability can be avoided by permuting the rows of the matrix. The runtime complexity for LU-decomposition is $O(n^3)$.

2.1.2 QR-decomposition Method

QR-decomposition [53] is based on a matrix decomposition where a non-singular square matrix A can be written as the product of $A = QR$, where R is an upper triangular and Q is an orthogonal one which satisfies $Q^T Q = I$, where Q^T is the transpose of Q and I is an identity matrix. QR factorization requires that a matrix must be reduced to zero. There are various methods that can be used for computing the QR-decomposition, the most common being Householder reflections.

When applied to a given matrix, a Householder matrix can zero all elements in a column of the matrix, situated below a given element. For the first column of matrix A the appropriate matrix Q_1 is evaluated, which zeroes all elements in the first column of A below the first element. Similarly Q_2 zeroes all elements in the second column below the second element and so on up to Q_{n-1}. Hence
$R = Q_{n-1} \cdots Q_1.A$
Since the Householder matrices are orthogonal it follows:
$Q = (Q_{n-1} \cdots Q_1)^{-1} = Q_1 \cdots Q_{n-1}$.
QR-decomposition is numerically stable without permuting the rows of the matrix The runtime complexity of QR-decomposition is similar to LU-decomposition, $i.e$, $O(n^3)$.

$$\begin{bmatrix} 4 & -1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 4 & -1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 4 & -1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 4 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 4 & -1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & -1 & 4 & -1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & -1 & 4 & -1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 & -1 & 4 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 4 & -1 & 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & -1 & 4 & -1 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & -1 & 4 & -1 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & -1 & 4 & 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 4 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & -1 & 4 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & -1 & 4 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & -1 & 4 \end{bmatrix} \cdot \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \\ u_6 \\ u_7 \\ u_8 \\ u_9 \\ u_{10} \\ u_{11} \\ u_{12} \\ u_{13} \\ u_{14} \\ u_{15} \\ u_{16} \end{bmatrix} = \begin{bmatrix} 0.08 \\ 0.16 \\ 0.36 \\ 1.64 \\ 0.16 \\ 0.0 \\ 0.0 \\ 1.0 \\ 0.36 \\ 0 \\ 0 \\ 1.0 \\ 1.64 \\ 1.0 \\ 1.0 \\ 2.0 \end{bmatrix}$$

Figure 2.1: Sparse matrix example

2.2 Indirect Methods

The approximate methods that provide solutions for systems of linear constraints starting from an initial estimate are known as iterative methods. Iterative methods are preferable for large sparse matrices (where the number of coefficients is $O(n)$) because they solve sparse systems without taking into account zeros.

Most of the research on iterative methods focuses on using them for solving linear systems of equalities and inequalities for sparse square matrices, for which the most important method is Successive Over-Relaxation (SOR). They start from an initial guess and improve the approximation until the absolute error is less than the predefined tolerance.

The following subsections summarize some basic indirect methods.

2.2.1 Jacobi Method

The Jacobi method [70] is a simple technique to solve linear systems of equations with absolute values in each row and column that are dominated by the diagonal element.
Suppose we have given system of n equations and n unknowns in the form

$$Ax = b, \tag{2.6}$$

we write the matrix A in the form

$$A = L + D + U, \tag{2.7}$$

then we have,

$$B_j = -D^{-1}(L + U) \tag{2.8}$$

where L is a lower part, D is a diagonal part and U is an upper part

$$L = \begin{pmatrix} 0 & \cdots & 0 & \\ a_{21} & \cdots & 0 & \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & \cdots a_{n,n-1} & 0 & \end{pmatrix}, D = \begin{pmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{pmatrix}$$

$$\text{, and} \quad U = \begin{pmatrix} 0 & a_{21} & \cdots & a_{1n} \\ 0 & \cdots & \cdots & \cdots a_{n-1,n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & \cdots & \cdots 0 \end{pmatrix}$$

Now the Jacobi technique has the form.

$$x^{k+1} = B_J x^k + b_J, \tag{2.9}$$

where B_j is an iteration matrix for Jacobi and b_j is the right hand side vector for Jacobi. This equation (2.9) is used for theoretical purposes. We can rewrite the equation 2.6 for ith term as follows. In practice this equation (2.10) is used for computation purposes.

$$x_i^{k+1} = \frac{1}{a_{ii}}\{b_i - \sum_{j \neq i}^{n} a_{ij} x_j^k\} \tag{2.10}$$

To start the iterative procedure for the Jacobi method one has to choose the initial estimate and then substitute the solution in the above equation. Iterations are repeated until the residual difference is less than the predefined tolerance.
The time complexity for Jacobi method is $O(k)$ where k is the iteration number [34] .

2.2.2 Gauss-Seidel Method/ Successive Over-Relaxation

The best-known indirect method for solving linear constraints is the Gauss-Seidel (GS) method [79]. Given a system of *m* equations and *n* variables of the form

$$Ax = b, \tag{2.11}$$

Similarly to Jacobi method, we can split the matrix A in the form

$$A = D + L + U \tag{2.12}$$

Then we have ,

$$B_{GS} = -(D + L)^{-1}U \tag{2.13}$$

where $D + L$ is the diagonal and lower parts and U is an upper part.

$$D + L = \begin{pmatrix} a_{11} & 0 & \cdots & 0 \\ a_{21} & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}, \text{and} \quad U = \begin{pmatrix} 0 & a_{21} & \cdots & a_{1n} \\ 0 & \cdots & \cdots & \cdots a_{n-1,n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & \cdots & \cdots 0 \end{pmatrix}$$

Now the GS method has the form

$$x^{k+1} = B_{GS}x^k + b_{GS}, \tag{2.14}$$

where B_{GS} is an iteration matrix for Gauss-Seidel and b_{GS} is the right hand side vector for Gauss-Seidel. This equation (2.14) is used for theoretical purposes. We can rewrite the equation (2.11) for the ith term as follows

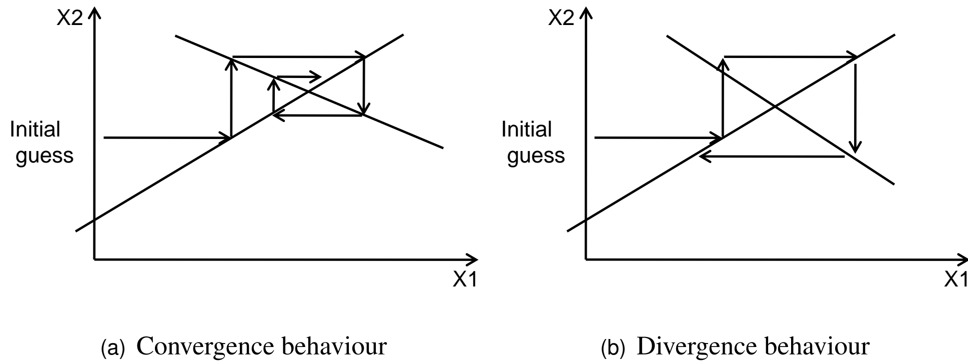
$$x_i^{k+1} = \frac{1}{a_{ii}}(b_i - \sum_{j=1}^{i-1} a_{ij}x_j). \tag{2.15}$$

In Equation (2.15), k is the iteration number and $i = (k \mod m) + 1$ (for deterministic Gauss-Seidel). This equation(2.15) is similar to the equation (2.10) in Jacobi method but the main difference is the linewise iteration. In Equation (2.15), the variable x_i, which is brought onto the left side, is called the *pivot variable*, and a_{ii} is the *pivot coefficient* or *pivot element* chosen for row i. An initial estimate for x is chosen, which usually does not fulfil the equations. The algorithm refines the estimate by repeatedly replacing all individual x_i so that the ith equation becomes fulfilled. This is done in round-robin fashion, and one full run through all n equations is one iteration, k being the iteration number. We can therefore write the process as:

$$x_i^{k+1} = \frac{1}{a_{ii}}(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{k+1} - \sum_{j=i+1}^{n} a_{ij}x_j^k) - x_i^k. \tag{2.16}$$

The algorithm iterates until the residual error is less than a pre-specified tolerance.

The Gauss-Seidel method is an improvement over the Jacobi method because it uses recently calculated components of the variable x whereas Jacobi uses previous values throughout the entire iteration. One iteration of Gauss-Seidel is equivalent to two Jacobi iterations but the complexity for GS is similar to Jacobi i.e. $O(k)$ because only constant term changes [34].



(a) Convergence behaviour (b) Divergence behaviour

Figure 2.2: Convergence and divergence behaviour of Gauss-Seidel

Convergence and divergence behaviour of Gauss-Seidel is shown in Figure 2.2.

Linear relaxation [112], also known as Successive Over-Relaxation (SOR), is an improvement of the Gauss-Seidel method. It is used to speed up the convergence of the Gauss-Seidel method by introducing a parameter ω, known as the relaxation parameter, so that

$x_j^{k+1} = x_j^k$ for $j \neq i = (k \mod m) + 1$

$$x_i^{k+1} = \frac{w}{a_{ii}}(b_i - \sum_{j=1}^{i-1} a_{ij}x_j - \sum_{j=i+1}^{n} a_{ij}x_j) + (-w)x_i^k. \tag{2.17}$$

This reduces to the Gauss-Seidel method if $\omega = 1$. $\omega > 1$ indicates over-relaxation and $\omega < 1$ under-relaxation.

2.2.3 Termination Criteria

Generally, the iterative method stops when the accuracy of the approximate solution is fulfilled. A good termination criterion is very useful for an iterative method because the iterative method should be stopped if the approximate solution is found. On the other hand, if the criterion is too severe then iterative method never stops and cannot find an approximate solution.

The termination criterion that we used for the above explained iterative methods was the difference between a computed iterate and the true solution of a linear system, measured in a vector norm described as follows.

$$\|r^{k+1}\| = \|b - Ax^i\|_2 \leq \epsilon \tag{2.18}$$

Thus the iterative method is terminated if the maximum of residual error is less than a pre-specified tolerance. Here, $\|r^{k+1}\|$ is a norm of the residual vector and b is a right hand side vector. A is the left hand side and ϵ is the prescribed tolerance. A good stopping criterion should identify when the error is small enough to stop. It should stop if the error is no longer

decreasing or decreasing too slowly, and limit the maximum amount of time spent iterating.

2.2.4 Advantages

Iterative methods such as SOR have certain advantages over direct methods. Iterative methods are typically simpler to implement, resulting in smaller programs. Furthermore, they have less round-off error than direct methods [5]. They start with an approximate answer and improve its accuracy in each iteration, so that the algorithm can be terminated once a sufficient accuracy is achieved.

Compared to direct methods, iterative methods are very efficient for sparse matrices, i.e. matrices where the number of non-zero elements is a small fraction of the total number of elements in the matrix. They are faster than direct methods because zero-coefficients are ignored implicitly, whereas direct methods have to process the zero-coefficients explicitly [5]. This implies that iterative methods need not to store zero-coefficients explicitly, which leads to less memory consumption than with direct methods [24]. Considering these advantages, it would be useful if the limitations of SOR could be overcome.

2.2.5 Convergence

The convergence of the Gauss-Seidel method can be characterized by two related but quite distinct approaches. The first approach, which is the best-known theorem in this domain, is based on the coefficient matrix.

Definition 1. A matrix is called strictly diagonally dominant if for all i

$$|a_{ii}| > \sum_{j \neq i} |a_{ij}|. \tag{2.19}$$

If a coefficient matrix is strictly diagonally dominant, the problem is guaranteed to converge [34]. However, this is only a sufficient condition, and a very strong condition that can be easily violated.

The notion of diagonal dominance naturally gives rise to a more general quantity that describes the influence of a variable.

Definition 2. The influence of the kth variable in the ith constraint is

$$\text{influence}_{ik} = \frac{|a_{ik}|}{\sum_j |a_{ij}|}. \tag{2.20}$$

All influences of variables in a constraint sum up to 1. If the constraints are normalized by dividing by the denominator above, then the absolutes of the coefficients of the variables are simply their influences.

The following is a proof of convergence for Gauss-Seidel based on strict diagonal dominance criteria.

Gauss-Seidel Convergence based on Strict Diagonal Dominance

Theorem 1. If the linear system $Ax = b$ has a coefficient matrix that is strictly diagonally dominant then the Gauss-Seidel method will converge to $\bar{x}$ for any choice of x_0 and any b[1]

Proof. Let $\overline{x} = [\overline{x}_i \ldots \overline{x}_n]^T$ be the exact solution of the system $Ax = b$. Then for all $i \in \{0 \ldots n\}$, we have

$$\bar{x}_i = \frac{1}{a_{ii}}\{b_i - \sum_{j \neq i} a_{ij}\bar{x}_j\} \tag{2.21}$$

The ith element of the ith row of the coefficient matrix is the row dominant element for variable x_i and serves as the pivot element.

According to the Gauss-Seidel method one iteration is performed with the equation

$$x_i^{k+1} = \frac{1}{a_{ii}}\{b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{k+1} - \sum_{j=i+1}^{n} a_{ij}x_j^{k}\}. \tag{2.22}$$

With equations (2.21) and (2.22) we can define the error of the approximation of the current iteration by

$$\epsilon_i^{k+i} = \bar{x}_i - x_i^{k+1}. \tag{2.23}$$

By substituting the values of equation (2.23) with equations (2.22) and (2.21), we get

$$\epsilon_i^{k+1} = \frac{1}{a_{ii}}\{b_i - \sum_{j \neq i} a_{ij}\bar{x}_j\} - \frac{1}{a_{ii}}\{b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{k+1} - \sum_{j=i+1}^{n} a_{ij}x_j^{k}\}. \tag{2.24}$$

After simplification, we get

$$\epsilon_i^{k+1} = \frac{-1}{a_{ii}}\{\sum_{j=1}^{i-1} a_{ij}(\bar{x}_j - x_j^{k+1}) + \sum_{j=i+1}^{n} a_{ij}(\bar{x}_j - x_j^{k})\}. \tag{2.25}$$

As we can see from the equation above the error of the current iteration depends on the error of the previous calculations of the variable values (strictly speaking the error of the variables whose constraint with its pivot element is less than i from the current iteration $(k+1)$ and the error of the variables whose constraint with its pivot element is greater than i from the previous iteration (k)).

[1] The proof of the convergence of the Gauss-Seidel method is based on the formulation of Maron's convergence proof of the Jacobi method in the case of square coefficient matrices [90]. We extended this formulation to the Gauss-Seidel method.

Hence we can write

$$\epsilon_i^{k+1} = \frac{-1}{a_{ii}}\{\sum_{j=1}^{i-1} a_{ij}\epsilon_j^{k+1} + \sum_{j=i+1}^{n} a_{ij}\epsilon_j^{k}\}. \tag{2.26}$$

Since in the end we are interested in finding the exact solution of $\overline{x}_i$ we have to reduce the absolute value of ϵ_i. Hence we can focus on the absolute values and change the equation accordingly:

$$|\epsilon_i^{k+1}| = \frac{-1}{a_{ii}}\{|\sum_{j=1}^{i-1} a_{ij}\epsilon_j^{k+1} + \sum_{j=i+1}^{n} a_{ij}\epsilon_j^{k}|\} \tag{2.27}$$

Because $|\epsilon_i^{k+1}|$ is defined over the sum of all errors of all remaining variables, it has to be smaller than the sum of all coefficients of the ith constraint. This value in turn is smaller than the product of the maximum error times the fraction of the biggest difference between the pivot coefficient and all remaining coefficients of this constraint.

$$|\epsilon_i^{k+1}| \le |\frac{1}{a_{ii}}|\sum_{j\neq i}|a_{ij}||\epsilon_{max}| \le \delta|\epsilon_{max}| \tag{2.28}$$

with

$$\delta = \max\{\frac{\sum_{j\neq 1}|a_{1j}|}{|a_{11}|}, \frac{\sum_{j\neq 2}|a_{2j}|}{|a_{22}|}, \ldots, \frac{\sum_{j\neq n}|a_{nj}|}{|a_{nn}|}\}. \tag{2.29}$$

From Equation (2.28) it is clear that the error of ϵ_i^{k+1} is smaller than the maximum error of the previous iterations by a factor of at least δ. The convergence will therefore be assured if $\delta < 1$. If $\delta < 1$ in each iteration the error becomes smaller for all x_i by a factor of at least δ. □

A proof for necessary and sufficient condition of Gauss-Seidel convergence is described as follows.

A Necessary and Sufficient Condition for Gauss-Seidel Convergence

The second approach to characterizing convergence is based on a derived matrix, the iteration matrix, and leads to a necessary and sufficient condition involving the spectral radius of the iteration matrix that also applies for SOR, not only Gauss-Seidel.

Definition 3 (Iteration Matrix)**.** The changes to the estimate x^k in every step of the Gauss-Seidel method for $Ax = 0$ are given by a linear function. The matrix $M_\omega(A)$ of this function with $x^{k+1} = M_\omega(A)x^k$ is called the iteration matrix of $Ax = 0$.

The Gauss-Seidel method converges for all initial values if and only if the spectral radius of the iteration matrix is smaller than one. If that condition is not fulfilled, the problem will diverge, except for some special initial values (such as the solutions itself). The smaller the spectral radius of the iteration matrix, the faster the Gauss-Seidel method converges.

An important characteristic of the Gauss-Seidel method with regard to convergence is translation invariance.

Lemma 1 (Translation Invariance)**.** Let the Gauss-Seidel method for $Ax = b$ converge to $\bar{x}$ starting with x^0. Then the Gauss-Seidel method for $Ay = 0$ starting with $y^0 = x^0 - \bar{x}$ will have the same convergence behaviour, i.e. for all j, k we have $y_j^k = x_j^k - \bar{x}_j$.

With Lemma 1 we can simplify the existing proof [107] of convergence based on the iteration matrix.

Theorem 2. The Gauss-Seidel method for $Ax = b$ converges[2] if the Iteration Matrix is non-singular and $M(A)$ has a spectral radius smaller than 1.

Proof. If the spectral radius is smaller than 1, then $M(A)^n$ converges to the matrix 0. Hence Gauss-Seidel for $Ax = 0$ converges to the correct solution 0. From translation invariance it follows that the Gauss-Seidel method for $Ax = b$ converges. □

Preconditioning

Preconditioning is important for the successful use of iterative methods. Preconditioning was first considered by Censor et al. [19, 27] for reducing the condition number in order to improve the convergence of an iterative process. The convergence of the Gauss-Seidel method depends strongly on the spectral radius of an iteration matrix. The Gauss-Seidel method shows best convergence if all eigenvalues of an iteration matrix are less than one. Underrelaxation has generally a better convergence behaviour than Gauss-Seidel, but convergence is usually not a problem for Gauss-Seidel in practice.

Preconditioners can be used to speed up the convergence of the Gauss-Seidel method but they do not guaranty its convergence. Good preconditioners for iterative methods are scaling algorithms [105] and bipartite matching algorithms [36]. These algorithms scale the infinity norm of both rows and columns in a matrix to 1 and permute large entries to the diagonal of a sparse matrix. There are usually well-conditioned coefficient matrices in a constraint-based UI layout problem for which the Gauss-Seidel method converges quickly if appropriate pivot elements are chosen.

2.3 Linear Programming

Linear programming [13] deals with the optimization (minimization or maximization) of an objective function that satisfies a set of constraints.

[2]We treat inequalities as equalities or ignore them if they are fulfilled.

A specification as a linear program is trivially in general more expressive than a specification as a system of linear equations and inequalities. The specification as a system of linear equations and inequalities is a special case of linear programming with the trivial objective function 0.

Definition 4. Linear Programming is a problem which can be formulated in standard form as:
Minimize $c^T x$
Subject to $Ax = b$,$x \geq 0$,
where $c^T x$ is a linear objective function.

$Ax = b$ is a set of linear constraints and $x \geq 0$ requires non-negativity conditions.

In the maximization case, minimizing $c^T x$ is equivalent to maximizing $-c^T x$. Inequality constraints are included because $A^{'}x \leq b$ or $A^{''}x \geq b$ is equivalent to $Ax = b$ by including slack and surplus variables as required.

Linear Programming is mostly used in constrained optimization. A large number of optimization problems are LPs having hundred of thousands of variables and thousands of constraints. With the recent advancement in computer technology these problems can be solved in practical amounts of time. The most common algorithm to solve linear programming problems is called the simplex method, which is described below.

2.3.1 Simplex Method

The simplex method [33] also known as the simplex technique or simplex algorithm was developed in 1947 by the American mathematician George B.Dantzig. It is an iterative method and makes use of Gauss-Jordan elimination techniques. It has the advantage of being universal, i.e. any linear model for which a solution exists can be solved by it. It is defined as an algebraic process for solving linear programming problems.

The simplex method is an iterative process that starts at a feasible corner point(normally the origin) and systematically moves from one feasible extreme point to another, until an optimal point is eventually reached.
The simplex method usually has two stages, called phase-I and phase-II.
In phase-I, the algorithm finds a basic feasible solution.
In phase-II, the algorithm searches for an optimal solution. In phase-I, slack variables(a slack variable is added to a constraint to turn an inequality into an equation) are introduced to find a value of the decision variables where all the constraints are satisfied. Once a basic feasible solution is found, the search for an optimal solution can start. In phase-II, the algorithm moves from one extreme point to another to find the optimal solution. The next extreme point will be chosen such that the search direction is in the steepest feasible direction. This process continues until the optimum solution is reached. The complexity for simplex is $O(n^3)$.

2.4 Summary

This chapter introduced some basic numerical methods for solving linear constraints. In the case of solving linear constraints we have presented (i) direct methods: LU-decomposition, QR-decomposition and (ii) the iterative methods: Jacobi, Gauss-Seidel, SOR and the linear programming method, the simplex method. Direct methods become impractical for a sparse, large linear systems in three-dimensional space. The complexity of the direct method becomes $O(n^3)$ and storage requirement can be up to $O(n^2)$ due to fill-in. whereas iterative methods are designed to solve large sparse linear systems of equations efficiently. The complexity of an iterative method is $O(n)$.

As large sparse problems are encountered in GUI layout we prefer iterative methods over direct methods. Linear programming method, simplex, is also an iterative method but it uses one Gauss-Jordan elimination step per iteration, i.e. using a direct method which makes them slower than other iterative methods. Our main goal of this chapter is to provide the reader with sufficient information on each algorithm that we used for comparison purposes with our proposed algorithms in this thesis.

3

Extending Successive Over-Relaxation for Non-Square Matrices

3.1 Introduction

Successive Over-Relaxation (SOR) is a common method for solving linear problems as they occur in the science and engineering. In contrast to direct methods such as Gaussian elimination or QR-factorization, SOR is inherently efficient for problems with sparse matrices as they are often encountered, for instance, in the application domain of constraint-based user interface (UI) layout.

One domain where sparse problems frequently occur is user interface (UI) layout. This chapter describes the common properties of this domain, and delineates the solving approaches that have been proposed for it. The contributions of this chapter were motivated by and were evaluated for the constraint-based UI layout problem.

One of the most common iterative methods used to solve sparse linear systems is SOR. Starting with an initial guess, SOR repeatedly iterates through the constraints of a linear specification, refining the solution until a sufficient precision is reached. For each constraint, it chooses a *pivot variable* and changes the value of that variable so that the constraint is satisfied. Despite its efficiency for sparse systems, SOR is currently not used for constraint-based UI layout, for the reasons explained in the following.

A common property of many linear problems including constraint-based UI layout is that the

matrices corresponding to these linear problems are non-square. For example, when specifying UI layout with linear constraints, there are generally more constraints than variables. This is a problem for the common SOR method, which assumes that the problem matrix is square and has a non-zero diagonal.

The problem for non-square matrices is that of *pivot assignment*, i.e. the choosing of a pivot variable for each constraint during solving. The standard SOR algorithms choose the pivot variable on the diagonal of the coefficient matrix. In case of square matrices with non-zero diagonals, this is an easy way to ensure that every constraint has a pivot variable, and that every variable is chosen once so that its value can be approximated. However, in the general case the diagonal approach has several problems:

1. Not every constraint contains an element on the diagonal of the problem matrix if there are more constraints than variables.

2. Diagonal elements may be zero, making them infeasible as pivot elements.

3. Diagonal elements may be small compared to the other elements on the same row of the matrix, making them a bad choice that may cause the solving process to diverge.

The standard SOR algorithms usually assume that a pivot assignment has been performed and that the chosen pivot elements are placed on the diagonal of the problem matrix. In square matrices this can always be achieved by simple matrix transformations. However, in the case of non-square matrices, the problem number 1 above cannot be mitigated this way.

In this chapter we describe how the SOR method can be extended to deal with the above-mentioned problems. We propose three pivot assignment algorithms that can be used with any problem matrix, regardless of its shape or diagonal elements. The first algorithm selects pivot elements pseudo-randomly. The second algorithm selects pivot elements deterministically by optimizing certain selection criteria. The third algorithm selects the best constraint for a variable and the best variable for a constraint. The problem of pivot assignment in the case of non-square matrices and the three algorithms is explained in detail in Section 3.3.

In GUI layout mixed systems occur that contain both equality and inequality constraints. We extend SOR for solving linear inequalities that is described in Section 3.2.

3.2 Inequalities

SOR supports both linear equalities and linear inequalities. Inequalities are handled similarly to equalities [2, 52, 94]: in each iteration, inequalities are ignored if they are satisfied, and otherwise treated as if they were equalities. However, there are potential practical problems, which are described below using the following definitions.

Definition 5. Mixed system: A system containing equalities as well as inequalities is called a mixed system.

Definition 6. Maximum equality subsystem: A subsystem that consists of all the equations in a system of linear constraints is called maximum equality subsystem.

A mixed system with a square coefficient matrix cannot have a unique solution because this is only possible if there is an equality for each variable. In a typical mixed system, as it occurs for instance in constraint-based UI layout specifications, the maximum equality subsystem is under-determined, i.e. there are fewer equalities than variables, and the whole system has more constraints than variables. This means that for typical mixed systems the standard SOR algorithm, which only works on a square matrix, is insufficient. To use SOR for such mixed systems we have to extend it, e.g. by applying the algorithms proposed in this chapter.

3.3 Non-Square Matrices

As pointed out in Section 3.2, mixed systems usually have a non-square matrix with more constraints than variables. Furthermore, they may have zero-coefficients on the diagonal. In some cases, they may also have more variables than constraints (under-determined). In all these cases, the standard SOR algorithm cannot be applied. In this section, we briefly present some related work on constraint problems with non-square coefficient matrices and propose three pivot assignment algorithms. All these three algorithms help to overcome the limitations of the standard SOR algorithm.

3.4 Related Work

Linear systems with non-square matrices are typically solved using direct methods, such as the QR-factorization method [34]. The QR-factorization method is used to solve linear systems of equations. Several methods can be used to compute QR-factorization, e.g. the Gram-Schmidt process, Householder transformations, or Givens rotations [34]. These methods require the calculation of a significant number of matrix norms, which makes them slower than other methods such as the normal equations method.

The normal equations method [59], which is also a direct method, is used to solve linear systems of the form $A^T Ax = A^T b$. It is fairly simple to program, but suffers from numerical instability when solving ill-conditioned problems. The condition of the normal equation matrix $A^T A$ is worse than that of the original matrix A. When an original matrix is converted into a normal equation matrix and a right hand-side vector, information can be lost. The normal equations method is the most common method despite the loss of information because it has been shown that its accuracy can be improved if iterative refinement is applied [45]. Some

iterative methods like Gauss-Seidel use normal equations to solve non-square linear systems of equations [34].

There are some iterative methods [10] that can be used to solve systems of linear equations that are over-determined. These methods include the simplex, the conjugate gradient and generalized minimal residual methods. They have some limitations that make them inapplicable for some problems, however, and these are described below.

The simplex algorithm [33] is a well-known method used to solve linear programming problems. It is an iterative method, but one linear solving step per iteration is required, which means this method cannot be faster than linear solving alone. It moves from one feasible corner point to another and continues iteration until an optimal solution is reached. The revised simplex method [117] is a variant of the simplex algorithm which is computationally more efficient for large sparse problems.

While the simplex algorithm tries to find an optimal solution according to an arbitrary linear objective function, there are other methods that try to find a least squares solution to an over-determined system of linear equations. These methods find a solution whose sum of squared errors is minimal. UI methods that work on squared errors exists such as [91] but there are also methods using absolute value [11] which is currently the basis for all layouts in Apple,s Cocoa layout engine.

The generalized minimal residual method [108] is considered the most efficient method for solving least squares problems. One of the shortcomings of this method is its instability and poor accuracy of the computed solution due to the possible high ill-conditioning of the normal equations system. This method is unstable because a non-square matrix is converted into a square matrix by applying normal equations.

The conjugate gradient method [62] can solve linear systems of equations if the matrix is symmetric and positive definite. However, it only works for well-conditioned problems as it cannot converge otherwise. Several methods for iteratively solving linear least squares problems – so called Krylov subspace methods – are surveyed in [53].

The Jacobi and Gauss-Seidel algorithms [35] are extended to solve non-square matrices in the least squares sense by applying a hierarchical identification principle and by introducing block matrix inner-products.

The numerical difficulties encountered for under-determined problems are the same for over-determined problems, however round-off errors accumulated in the under-determined case are more complicated than in the over-determined case because the solution is not unique.

3.5 Pivot Assignment

Since the diagonal elements do not lend themselves naturally as pivot elements if the matrix is non-square, we need to explicitly select a pivot element for each constraint. In other words, we

need to determine a *pivot assignment*. Pivot assignment is also important for square matrices as it has an effect on convergence.

Definition 7. A pivot assignment is an assignment of constraints to variables

$$\gamma\colon \text{Constraints} \to \text{Variables}.$$

A *feasible* pivot assignment γ must be *surjective* and *total*. Surjectiveness is necessary because we require one constraint for each variable, otherwise the variable's value would not be changed by the algorithm. Totality is inherent in the definition of the SOR algorithm, which requires a pivot variable for every constraint.

3.5.1 Pivot Assignment Algorithms

Below we propose three pivot assignment algorithms, one random, one deterministic and two phase. While the random algorithm avoids the issues of surjectiveness and totality by randomization, deterministic and two phase algorithms ensure these properties, using the notions of most influential variables and constraints defined as follows.

Definition 8. The most influential variable of a constraint is the one with the highest influence. The most influential constraint of a variable is the constraint where the variable has the highest influence.

Random Pivot Assignment

The algorithm for random pivot assignment is given below as Algorithm 1. The random algorithm assigns the pivot variable for each constraint randomly in each iteration (line 2). This means that in general the pivot assignment is changed in each iteration.

It is not inherently obvious that randomized assignments work for the SOR approach, but it is the simplest approach that may work. Although the random algorithm generally does not make the optimal assignment with regard to convergence, it reduces the effect of bad assignments while allowing for good assignments. In particular, it is guaranteed that every suitable variable will be chosen as pivot variable at some point. The general assumption underlying randomized algorithms is that the effect of good choices outweighs the effect of bad choices.

One of the drawbacks of random assignment is that it causes fluctuation in the error. This makes it harder to recognize whether the algorithm diverges, or whether fluctuations are only temporary. To address this problem, we propose a deterministic approach in the following section.

Input: Constraints (**C**)
Output: Pivot Assignment γ

1: **for** each constraint c **do**
2: Choose variable x of c randomly
3: Assign $\gamma(c) = x$
4: **end for**

Algorithm 1: Random pivot assignment

Deterministic Pivot Assignment

The algorithm for deterministic pivot assignment is given below in Algorithm 2. It creates a single pivot assignment that is used consistently during the solving process and is explained in the course of the following proof of correctness.

Theorem 3. The deterministic algorithm produces only feasible assignments.

Proof. In lines 1–7 each constraint is assigned a variable x, therefore the resulting assignment is total. In lines 8–11 every variable y that has not yet been assigned a constraint is assigned a new constraint, which is a duplicate of an existing constraint. As a result, the resulting assignment is surjective.

If the matrix is diagonally dominant, at the time the algorithm iterates over a particular constraint, the most influential variable of this constraint will still be unassigned. After the first loop, there will be no unassigned variables left. As a result, the algorithm chooses the diagonal elements as pivots in the case of diagonally dominant matrices. Duplicating constraints of a problem does not change the problem, hence this is a valid transformation. □

Input: Constraints (**C**)
Output: Pivot Assignment γ

1: **for** each constraint c **do**
2: **if** some variables of c are still unassigned **then**
3: Choose unassigned variable x of c with the largest influence, assign $\gamma(c) = x$
4: **else**
5: Choose the most influential variable x of c, assign $\gamma(c) = x$
6: **end if**
7: **end for**
8: **for** each still unassigned variable y **do**
9: Find the most influential constraint c for y
10: Duplicate c to c', assign $\gamma(c') = y$
11: **end for**

Algorithm 2: Deterministic pivot assignment

Two Phase Pivot Assignment

The algorithm for two phase pivot assignment is given below in Algorithm 3. This algorithm tries to find the most influential constraint for a variable and the most influential variable for a constraint.

Input: Constraints (**C**), variables (**X**)
Output: Pivot Assignment γ

1: **for** all constraints C **do**
2: Choose the most influential variable x of constraint c, assign $\gamma(c) = x$
3: Remove x from X
4: **end for**
5: **for** all variables $x : X$ **do**
6: Choose the most influential constraint c for x
7: Duplicate c to c', assign $\gamma(c') = x$
8: **end for**

Algorithm 3: Two phase pivot assignment

In lines 1–3 it tries to assign a best variable x for all constraints. In lines 4–7 for all variables X, it tries to assign a best constraint x , which is a duplicate of an existing constraint. This algorithm constantly selects the pivots which have maximum impact. But with that some variables are not used as pivots or some constraints are used several times with different pivots. Therefore, the resulting assignments are not total and surjective.

In the general case constraint-based UIs are over-determined, which can result in conflicts between constraints of the problem. A proper pivot assignment algorithm alone is not sufficient to deal with such cases. A technique to handle conflicts between constraints, e.g. in the form of soft constraints, is required. We describe techniques to handle conflicts between constraints in Chapter 4, which also experimentally evaluates our proposed pivot assignment algorithms together with the conflict resolution algorithms with regard to their convergence and performance.

3.6 Summary

We have proposed new algorithms for using SOR for solving constraint-based UI layout problems. In particular, we presented the algorithms for pivot assignment that make it possible to solve problems with non-square coefficient matrices. We also extend SOR for solving linear inequality constraints.

4

Extending Successive Over-Relaxation for Soft Constraints

4.1 Introduction

Besides its inability to deal with non-square matrices, the common SOR method has another shortcoming. Many problems, such as constraint-based UI layout, may contain conflicting constraints. This may be caused by over-constraining, i.e. by adding too many constraints, making a problem infeasible. If a specification contains conflicting constraints, the common SOR method simply will not converge.

To deal with conflicts, *soft constraints* can be introduced. One simple example of constraint-based UI layout for hard and soft constraints is shown in Figure 4.1.

In contrast to the usual *hard* constraints, which cannot be violated, soft constraints may be violated as much as necessary if no other solution can be found. Soft constraints can be prioritized so that in a conflict between two soft constraints only the soft constraint with the lower priority is violated. This leads naturally to the notion of *constraint hierarchies*, where all constraints are essentially soft constraints, and the constraints that are considered "hard" simply have the highest priorities [21]. Using only soft constraints has the advantage that a problem is always solvable, which cannot be guaranteed if hard constraints are used.

In this chapter we propose three conflict resolution algorithms for solving systems of prioritized linear constraints with the SOR method. The first algorithm successively adds non-

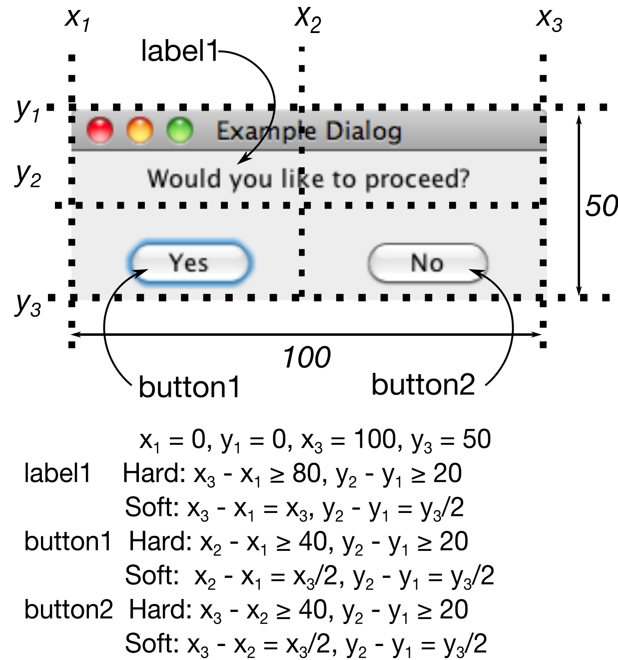


Figure 4.1: Example of constraint-based UI layout with hard and soft constraints

conflicting constraints in descending order of priority. The second algorithm starts with all constraints and successively removes conflicting constraints in ascending order of priority. The third algorithm is a mixture of both and adapts the binary search algorithm to the problem of searching for the best conflict-free subproblem. These algorithms yield conflict-free subproblems to a given problem. There are some existing algorithms for finding feasible subproblems for sets of constraints [29], but they do not take into account prioritization of constraints, which is important for constraint-based UI layout problems.

The pivot assignment algorithms presented in Chapter 3 and the conflict resolution algorithms presented below give a total of six different solution procedures that enable SOR to be applied to more linear constraint problems, such as constraint-based UI layout. These solution procedures were experimentally evaluated with regard to their convergence and performance, using randomly generated UI layout specifications. The results show that most of the proposed algorithms are optimal and efficient. Furthermore, we observed that some of our implemented solvers outperform Matlab's LINPROG linear optimization package [115], LP-Solve [17] and the implementation of QR-decomposition of the Apache Commons Math Library [8]. LP-Solve is a well-known linear programming solver that has been used for constraint-based UI layout. The implementation of QR-decomposition in the Apache Commons Math Library is an example of a direct method. The methodology as well as the results of the evaluation can be found in Section 4.6. Section 4.7 presents our conclusions.

For sparse linear problems, SOR is known to perform very well. However, SOR in its standard form cannot be applied to the constraint-based UI layout problems for two reasons. First, the coefficient matrices are non-square: there are usually more constraints than variables.

Second, many of the constraints are soft because they describe desirable properties in the layout (e.g. preferred sizes), which cannot be satisfied under all conditions (e.g. all layout sizes). As a result, the existing UI layout solvers use algorithms other than SOR. Some of these solvers are discussed in Section 4.2.

Hard constraints are constraints that must always be satisfied. If this is impossible, there is no solution. For many problems, including constraint-based UI layout, conflicting constraints occur naturally in specifications, as they express properties of a solution that are desirable but not mandatory. As a result, soft constraints need to be supported that are satisfied if possible, but do not render the specification infeasible if they are not. A natural way to support soft constraints is to treat all constraints as soft constraints, with different priorities (p). These priorities can be defined as a total order on all constraints that specifies which one of two constraints should be violated in case of a conflict.

To define the solution of a system of prioritized soft constraints, we first have to define the subset $E \subseteq$ *Constraints* of *enabled constraints*. We consider the characteristic function $\mathbf{1}_E$: *Constraints* $\rightarrow \{0, 1\}$ of E, which expresses whether a constraint is contained in E, to construct an integer in binary representation (ι). According to their priority, each constraint is represented by a bit of that integer, with constraints of higher priority taking the more significant bits. The value of the characteristic function for the constraint with the highest priority is considered the most significant bit. Then such subsets can be compared by using the numerical order $\geq$ of the integers. We are interested in the subset that is largest in that order and still fulfills the following property: all constraints in the subset are non-conflicting.

Below we first discuss existing approaches for solving linear soft constraints. Then, we describe three algorithms that address support for soft constraints in the SOR method: prioritized irreducible infeasible subsystem (IIS) detection, prioritized deletion filtering and prioritized grouping constraints.

4.2 Related Work

All constraint solvers for UI layout must support over-determined systems. The commonly used techniques for dealing with over-determined problems are weighted constraints and constraint hierarchies [47,63,92]. Weighted constraints are typically used with some general forms of direct methods, while constraint hierarchies are especially utilized in linear programming based algorithms. Many UI layout solvers are based on linear programming and support soft constraints using slack variables in the objective function [11,13,22,86,91].

Most of the direct methods for soft constraint problems are least squares methods such as LU-decomposition and QR-decomposition [125]. The UI layout solver HiRise [65] is an example of this category. Its successor, HiRise2 [66] solves hierarchies of linear constraints by applying an LU-decomposition-based simplex method.

However, if weights are used to express a hierarchy of constraints the differences between them has to be very large. This in turn can push numerical limits. Hence it is desirable to enforce priorities of constraints without using weights directly.

Many different local propagation algorithms have been proposed for solving constraint hierarchies in UI layouts. The DeltaBlue [46], SkyBlue [110] and Detail [67] algorithms are examples of this category. The DeltaBlue and SkyBlue algorithms cannot handle simultaneous constraints that depend on each other. However, the Detail algorithm can solve constraints simultaneously based on local propagation. All of the methods for handling soft constraints utilized in these solvers are designed to work with direct methods, so they inherit the problems that direct methods usually have with sparse matrices.

QUICKXPLAIN [77] tries to find a conflict-free constraint system by successively adding or removing constraints from the group of constraints which is similar to prioritized grouping constraints. The groups of constraints to be added or removed are determined by a recursive decomposition of the problem. It is mixture of QuickSort and MergeSort in the sense that QUICKXPLAIN must solve both of the two subproblems resulting from the problem decomposition. In contrast to QuickSort and MergeSort, the result of the right subproblem, which is solved first by QUICKXPLAIN, has an impact on the formulation of the left subproblem. This is a particularity of QUICKXPLAIN.

The Maximum Satisfiability (MAXSAT) problem is a generalization of the Satisfiability (SAT) problem which can represent optimization problems. The SAT problem tries to find an assignment that satisfies all the constraints if one exists; otherwise no satisfiable assignment exists. The goal of MAXSAT is to find an assignment that satisfies the maximum number of constraints. The MAXSAT solvers are based on branch and bound solvers and satisability testing [6, 7, 18, 60, 83, 84, 89].

The problem of finding the largest possible subset of constraints that has a feasible solution given a set of linear constraints is widely known as the Maximum Feasible Subsystem (MaxFS) problem [29]. The dual problem of MaxFS is the problem of finding the irreducible infeasible subsystem (IIS) [3]. If one more constraint is removed from an IIS, the subsystem will become feasible. For both problems different solving methods were proposed.

To solve the MaxFS problem, non-deterministic as well as deterministic methods were proposed. Some of these methods use heuristics [4, 88]. Only a few methods solve the problem deterministically, among which is the branch and cut method proposed by Pfetsch [99] is such a deterministic method. Besides methods to solve MaxFS, there are methods to solve the IIS problem, including deletion filtering, IIS detection algorithm, and the grouping constraints method.

Deletion filtering [30] starts with the set of all constraints. For each constraint in the set, the method temporarily drops the constraint from the set and checks the feasibility of the reduced set. If the reduced set is feasible, the method returns the dropped constraint to the set of constraints. If the reduced set is infeasible, the method removes the dropped constraint per-

manently. Baker et al. [12] proposed the algorithm, Diagnosis of Over-determined Constraint Satisfaction Problems. This algorithm tries to find the set of least important constraints that can be removed to solve the remaining constraint satisfaction problem. If the solution is not optimal then it tries to find next-best sets of least-important constraints until an optimal solution is found.

The IIS detection algorithm [118, 119] starts with a single constraint and adds constraints successively. If the system is infeasible after adding a new constraint, then the method discards the new constraint.

The grouping constraints method was introduced by Guieu and Chinneck [56] to speed up the IIS detection algorithm and deletion filtering. It adds or drops constraints in groups by using the deletion filtering or IIS detection algorithms.

Even though these methods deal with the problem of finding a feasible subsystem, it is not possible to apply them directly. The main reason for this is that they do not consider prioritized constraints, as necessary for problems such as constraint-based UI layout. As discussed in Section 4.1, we have to find not only the set with the maximum number of constraints, but also the set of constraints with $max(\iota)$. We call this problem *prioritized MaxFS*, and propose as a solution the following algorithms: prioritized IIS detection, prioritized deletion filtering and prioritized grouping constraints.

4.3 Prioritized IIS Detection

In Prioritized IIS detection, which is depicted as Algorithm 4, we start with an empty set E of enabled constraints (line 1). We add constraints incrementally in order of descending priority so that E is conflict-free, until all non-conflicting constraints have been added. Iterating through the constraints, we add each constraint tentatively to E ("enabling" it), and try to solve the resulting specification (line 7). Note that whenever a constraint is added, the pivot assignment needs to be recalculated. If a solution is found, we proceed to the next constraint. If no solution is found, the tentatively added constraint is removed again. In that case, the previous solution is restored and we proceed to the next constraint. This algorithm assumes that the method used for solving converges if there is no conflict. The algorithm approximates $max(\iota)$ starting from the most significant bit and progressing down to the least significant bit. This property distinguishes our algorithm from the existing IIS detection algorithm.

4.4 Prioritized Deletion Filtering

Prioritized deletion filtering is an algorithm that assumes a predicate $conflicting(c)$ with certain properties to exist. The assumption is that during one unsuccessful attempt to solve the current

Input: Constraints (**C**)
Output: Non-conflicting constraints

1: DISABLE(**C**)
2: SORT(**C**) (by priority)
3: **for** each constraint c in order of priority, descending **do**
4: Remember current variable values
5: ENABLE(c)
6: Assign pivot elements for all constraints
7: Apply SOR
8: **if** solution not optimal **then**
9: DISABLE(c)
10: Restore old variable values
11: **end if**
12: **end for**

Algorithm 4: Prioritized IIS Detection

specification, we can collect reliable information on each single constraint as to whether it is conflicting.

The steps are shown in Algorithm 5. We start with all constraints enabled, i.e. $E =$ *Constraints* (line 1). We try to solve the specification, and if an optimal solution is found, this means E is conflict-free. In this case, we return the solution. Otherwise, we remove the conflicting constraint with the lowest priority from E ("disabling") and recalculate the pivot assignment.

With a very simple heuristic predicate based on the error fluctuation, as described below in detail, this algorithm was used quite successfully during our evaluation. However, even if one assumes a completely reliable predicate $conflicting(C)$, the set of constraints getting disabled is generally larger than allowed in our definition of soft constraints. In contrast to prioritized IIS detection, if there is a conflicting constraint c of a higher priority in a specification, that constraint will only be deleted after it might have already triggered removal of a constraint d of a lower priority. After c is removed, d might be solvable, but has already been lost. We present this approach to provide another perspective on addressing soft constraints, and as a motivation for our third algorithm which is described in Section 4.5.

Currently, we use the following heuristic: $conflicting(c)$ is true if the value of its pivot variable $\gamma(c)$ has been changed significantly during the last SOR iteration. While this condition is true for conflicting constraints, it is not a sufficient condition, as other non-conflicting constraints may be affected by a conflict and hence satisfy this condition, too.

Input: Constraints ($\mathbf{C}$)
Output: Non-conflicting constraints

```
1:  ENABLE(C)
2:  SORT(C) (by priority)
3:  for each constraint do
4:     Assign pivot elements for all constraints
5:     Apply SOR
6:     if solution is optimal then
7:        return solution
8:     end if
9:     for each constraint c in order of priority, ascending do
10:       if conflicting(c) then
11:          DISABLE(c)
12:          break
13:       end if
14:    end for
15: end for
```

Algorithm 5: Prioritized Deletion Filtering

4.5 Prioritized Grouping Constraints

The prioritized grouping constraints algorithm is a combination of prioritized IIS detection and prioritized deletion filtering algorithms. It tries to find a conflict-free constraint system by successively adding or removing constraints from the system of constraints. If a constraint system is conflict-free, the algorithm adds constraints; if it has conflicts, the algorithm removes constraints. It adds and removes not only one constraint at a time, but also groups of constraints, and the size of the groups follows the classic patterns of a binary search approach. The algorithm ends if the system is conflict-free and no more constraints can be added. This algorithm, in contrast to the prioritized deletion filter, does not require a predicate $conflicting(c)$.

The steps are shown in Algorithm 6 below. First, the algorithm is initialized by sorting the list of constraints ($\mathbf{C}$) (line 1) and initializing some variables (line 2). The variables $beginning$ and end determine the upper-inclusive and the lower-exclusive bounds of the area of the list of constraints which possibly contains conflicting constraints, e.g. the *s*earch window. These bounds are adjusted while the algorithm is running.

When initializing the algorithm, we set $end = 2|\mathbf{C}|$, which is adjusted to $end = |\mathbf{C}|$ in the first iteration. Hence in the second iteration we check the whole list from the first entry ($beginning = 0$) to the last entry ($end = |\mathbf{C}| - 1$).

After initialization the algorithm enters a loop which iteratively finds the prioritized MaxFS ($max(\iota)$). First, the algorithm checks whether the calculated solution of the previous step is optimal. If this is the case, we know that, at least up to end, there is no conflict in the constraint list. We can, therefore, set the upper bound $beginning$ of the search window to end, ignore the old search window in the following iterations, and increase the lower bound end to form a new

window. The variable value end is increased exponentially with δ.

If the solution is not optimal, we have either identified a conflicting constraint, or we still need to decrease the size of the search window exponentially (line 15). We have found a conflicting constraint if the size of the search window is shrunk to a single constraint (line 10). In that case, we deactivate this constraint (line 11), move the upper bound of our search space to index position end (line 12), and set the size of our search window to one (line 13) for the following iteration.

Now, after the bounds of the search window are calculated, the constraints within the search window $(beginning, end)$ are enabled and all constraints below the search window are disabled (line 18). The constraints above $beginning$ are still enabled from the preceding iterations. Finally, the pivot elements for the enabled constraints are determined and the problem is solved for the enabled constraints (lines 19 and 20).

Similar to the aforementioned algorithms, this algorithm assumes that SOR converges if there is no conflict in the system. Under that assumption, it finds $\max(\iota)$, i.e. it will end with the same constraints as prioritized IIS detection. However, in contrast to prioritized IIS detection, it adds and removes constraints in bigger steps, which reduces the number of required iterations.

Input: Constraints ($\mathbf{C}$)
Output: Non-conflicting constraints

```
1:  SORT(C) by priority
2:  δ ← 1; beginning ← 0; end ← 2(|C|)
3:  while beginning < |C| do
4:     if solution optimal then
5:        Remember current variable values
6:        beginning ← end
7:        end ← end + δ (or |C| if it is out of bounds); δ ← 2δ
8:     else
9:        Restore old variable values
10:       if end = beginning + 1 then
11:          DISABLE(C[beginning])
12:          beginning ← end
13:          end ← end + 1 (or |C| if it is out of bounds); δ ← 2
14:       else
15:          end ← beginning + (end−beginning)/2
16:       end if
17:    end if
18:    ENABLE(C[beginning ... (end − 1)]) and DISABLE(C[end ... (|C| − 1)])
19:    Assign pivot elements for all enabled constraints
20:    SOLVE enabled constraints
21: end while
```

Algorithm 6: Prioritized Grouping Constraints

Figure 4.2 depicts a run of the prioritized grouping constraints algorithm. In this example,

Figure 4.2: Example run of the prioritized grouping constraints algorithm

the constraint problem consists of 10 constraints, ordered according to their priority. The constraint pairs (2, 9), (5, 8) and (7, 10) are conflicting. The objective is to find a problem with $max(\iota)$.

The algorithm starts in iteration 0 by just initializing the size of the search window and solving the complete list for the first time. With that result it enters iteration 1 with a search window that consists of the complete list of constraints (0 – 9), and tries to solve them. This problem cannot be solved since we have three conflicts in the list, as described above. In iteration 2, the search window is halved and the algorithm tries to solve the problem in the upper part (0 – 4). This subproblem is solvable and the algorithm moves the beginning of the search window to index 5 and starts with a new search window of size $\delta = 2$ in iteration 3. The new subproblem is solvable as well and δ is doubled to 4 so that the subproblem in iteration 4 contains again the whole list of constraints which are not solvable. In iteration 5 the search window is halved again. Since only one constraint is in the search window left and the problem is still not solvable the problem in the search window must be conflicting with one of the higher prioritized constraints and has to be disabled. The new search window in iteration 6 starts below the disabled constraint with size 2. It again contains conflicting constraints and is halved which yields a search window of size 1 in the next iteration and a subproblem which is still not solvable. Hence constraint 8 is conflicting as well and is disabled. Again the search window is moved below the disabled constraint in iteration 8. The new subproblem is not solvable and constraint 9 is disabled. In the last iteration the search window is of size 0 and the problem is solvable, which indicates that all conflicts are found and the algorithm can terminate. As the example shows, the prioritized grouping constraints algorithm deactivates only lower prioritized conflicting constraints resulting in $max(\iota)$. This property distinguishes our algorithm from the existing grouping constraints algorithm.

4.6 Experimental Evaluation

In this section, we present an experimental evaluation of the proposed algorithms. We conducted three different experiments to evaluate (1) their convergence behaviour, (2) their performance in terms of computation time, and (3) their ability to detect and resolve conflicts (the quality of a solution).

4.6.1 Methodology

For all three experiments we used the same computer and test data generator, but instrumentalized the algorithms differently. We used the following setup: a desktop computer with an Intel i5 3.3GHz processor and 64-bit Windows 7 running an Oracle Java virtual machine. Layout specifications were randomly generated using the test data generator described in Algorithm 7. For each experiment the same set of test data was used. The specification size was varied from 4 to 2402 constraints, in increments of 4 (2 new constraints for the position and 2 new constraint for the preferred size of a new widget). For each size, 10 different layouts were generated, resulting in a total of 6000 different layout specifications being evaluated. A linear relaxation parameter of 0.7 and a tolerance of 0.01 were used for SOR. We use 1000 maximum number of iterations until the algorithm gets near optimal solutions or an indication of likely infeasibility of the system.

In Experiment 1, we investigated the convergence behaviour of each algorithm by measuring the number of sub-optimal solutions. A solution is sub-optimal if the error of a constraint (the difference between the right-hand and left-hand sides) is not smaller than the tolerance.

In Experiment 2, we measured the performance in terms of computational time T in milliseconds (ms), depending on the problem size measured in number of constraints c. Each of the proposed algorithms was used to solve each of the problems of the test data set, and the time was taken. As a reference, all the generated specifications were also solved with Matlab's LINPROG solver [115] and LP-Solve [17]. We selected these two solvers because LINPROG is widely known for its speed [1], and LP-Solve has previously been used to solve constraint-based UI layout problems [86]. Additionally, we wanted to know how well our algorithms could compete with a direct method. Hence we also used the implementation of QR-decomposition in the Apache Commons Mathematics Library [8], which is a freely available open-source library.

In Experiment 3, we evaluated the quality of the solutions, which is given by the integer ι. As explained in Section 4.1, the algorithms should find $\max(\iota)$, so a solution with a larger ι has a better quality. We consider $\bar{\iota}$, the bit-wise negation of ι, as this allows us to differentiate between solutions of different quality more easily. The ι values of solutions for a problem differ usually only in the less significant bits (the most important constraints are usually enabled), whereas $\bar{\iota}$ reflects noteworthy quality differences in the more significant bits.

[1] http://plato.asu.edu/ftp/lpfree.html

To calculate $\bar{\iota}$, we set a bit in a bit array of the length of the list of constraints if a constraint is disabled. So in the worst case, $\bar{\iota}$ can become as large as 2^{2402}. Since such numbers are hard to evaluate and interpret, we simplify them by only expressing an ordinal relationship between the solutions of all 8 algorithms for one problem. This is done with integer values $rank$, expressing the ranking of a solution: one is the rank of the solution with the lowest $\bar{\iota}$, and 6 is the rank of the solution with the highest $\bar{\iota}$ for a given problem. Thus, the higher the rank the worse the solution of a solver. In the case of a ties, i.e. if two or more algorithms produce the same $\bar{\iota}$, we use the mean of the involved ranks, as is usually done in such situations to preserve the sum of all ranks.

Finally, we use the rank values to compare all algorithms pairwise. We compare two algorithms x and y by testing the distribution of the differences (d)

$$d = rank_x - rank_y$$

for all problems with a *Wilcoxon signed-rank* test on a significance level of $\alpha = 0.001$. If the test accepts the alternative hypothesis that $d < 0$, we conclude that algorithm x produces better results than algorithm y. These results are aggregated over all comparisons. We do not consider QR decomposition in this experiment because it just finds an unweighted least squares solution, i.e. without considering any priorities.

Algorithm 7 shows how random sets of areas A are created by partitioning the bounding area of a GUI. First, A only contains the bounding area of the GUI. Then, while the number of areas $|A|$ in A is less than the number of areas n_{areas} that the layout should contain, we divide one of the existing areas, thus increasing the total number of areas by one. Line 4 randomly chooses an area a of A, and line 5 removes it from A. Then, we randomly decide whether to divide a vertically or horizontally. Random is a random value between 0 and 1. x_{new} is a new x-tab that is inserted in order to divide a vertically; y_{new} is a new y-tab that is inserted in order to divide a horizontally. Because of the growing number of smaller areas and the uniformly random choice of the area that is subdivided next, the algorithm produces layouts with some large areas and many small ones. In our tests, we put a button into each of the areas of the generated layouts. The total GUI size was chosen randomly to be (800 100, 600 100). A random minimum size was set for each area to be (10 (400/n_{areas}), 10 (300/n_{areas}). Minimum sizes are important for the controls in the areas to be rendered correctly, and it is important to reduce the maximum for each minimum size with increasing $nareas$ in order to have a feasible layout. If the minimum sizes are too large then the minimum widths or minimum heights of adjacent areas might add up to more than the total size of the GUI so that there cannot be a solution. For each area, p_{expand} and p_{shrink} are chosen randomly between 0 and 1.

1: **function** GENERATE(n_{areas})
2: A $\leftarrow (left, top, right, bottom)$
3: **while** $|A| < n_{areas}$ **do**
4: a $\leftarrow randomElement(A)$
5: $A \leftarrow A - (a)$
6: **if** $random < 0.5$ **then**
7: $A \leftarrow AU(a.left, a.top, x_{new}, a.bottom), (x_{new}, a.top, a.right, a.bottom)$
8: **else**
9: $A \leftarrow AU(a.left, a.top, a.right, y_{new}), (x.left, y_{new}, a.right, a.bottom)$
10: **end if**
11: **end while**
12: **end function**

Algorithm 7: Generation of a random partition of areas.

4.6.2 Results

In Experiment 1, we found that all algorithms converge. This result is obvious since the algorithms are designed to find a solvable subproblem.

In Experiment 2, we analyzed the trends of the computational performance of the algorithms using different regression models (linear, quadratic, cubic and log). We found that the best-fitting model is the polynomial model

$$T = \beta_0 + \beta_1 c + \beta_2 c^2 + \beta_3 c^3 + \epsilon.$$

Key parameters of the models are depicted in Table 4.2; a graphical representation of the models can be found in Figures 4.3 – 4.5. Table 4.1 explains the symbols used.

Table 4.1: Symbols used for the performance regression model

Symbol	Explanation
β_0	Intercept of the regression model
β_{1-3}	Estimated model parameters
c	Number of constraints
T	Measured time in milliseconds
R^2	Coefficient of determination of the estimated regression models

For some strategies, some parameters do not have a significant effect, which can be interpreted as the complexity of the algorithm not following a certain polynomial trend. For example, prioritized deletion filtering with deterministic pivot assignment seems to have a purely quadratic runtime behaviour. For a better comparison of the runtime behaviour of the strategies, we considered all combinations of the soft constraint and pivot assignment algorithms. Figure 4.3 illustrates the performance comparison of prioritized IIS detection, prioritized deletion filtering and prioritized grouping constraints using random pivot assignment. As the graphs

Table 4.2: Regression models for the different solving strategies

Strategy	β_0	β_1	β_2	β_3	R^2
Pr. grouping constraints / deter.	1283^{***}	-10.45^{***}	$2.350 \cdot 10^{-02***}$	$5.865 \cdot 10^{-06***}$	0.9877
Pr. grouping constraints / rand.	8.341^{***}	$-4.692 \cdot 10^{-02***}$	$1.225 \cdot 10^{-04***}$	$-1.305 \cdot 10^{-08***}$	0.9917
Pr. deletion filtering / rand.	-0.6399	$5.333 \cdot 10^{-03}$	$8.946 \cdot 10^{-05***}$	$2.831 \cdot 10^{-09***}$	0.9925
Pr. IIS detection / rand.	4.174^{***}	$-2.270 \cdot 10^{-02***}$	$1.620 \cdot 10^{-04***}$	$-1.087 \cdot 10^{-08***}$	0.9957
Pr. IIS detection / deter.	-619.5^{***}	3.953^{***}	$-4.368 \cdot 10^{-03***}$	$1.243 \cdot 10^{-05***}$	0.9971
Pr. deletion filtering / deter.	1.537	$-7.698 \cdot 10^{-03}$	$1.668 \cdot 10^{-04***}$	$7.015 \cdot 10^{-10}$	0.9893
LINPROG	18.29^{***}	$1.591 \cdot 10^{-04}$	$4.934 \cdot 10^{-05***}$	$1.577 \cdot 10^{-08***}$	0.9367
LP-Solve	-2.491^{***}	$3.924 \cdot 10^{-02***}$	$2.079 \cdot 10^{-04***}$	$1.904 \cdot 10^{-08***}$	0.9900
QR-Decomposition	-37.70^{***}	0.2802^{***}	$-4.009 \cdot 10^{-04***}$	$2.850 \cdot 10^{-07***}$	0.9989

Significance codes: *** $p < 0.001$, deter: deterministic , rand: random

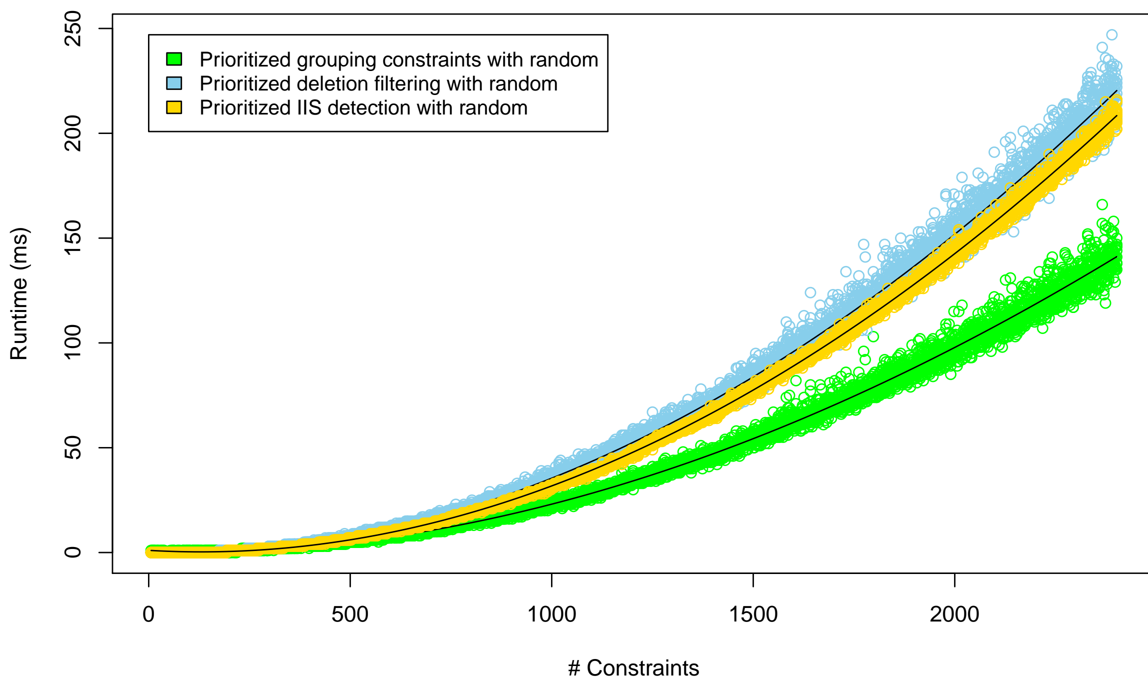


Figure 4.3: Performance comparison of prioritized IIS detection, prioritized deletion filtering and prioritized grouping constraints using random pivot assignment

indicate, prioritized grouping constraints exhibits better performance than prioritized deletion filtering and prioritized IIS detection.

Figure 4.4 compares prioritized IIS detection, prioritized deletion filtering and prioritized grouping constraints using deterministic pivot assignment. Generally, these strategies are slower than the strategies with random pivot assignment because the computation of the pivot assign-

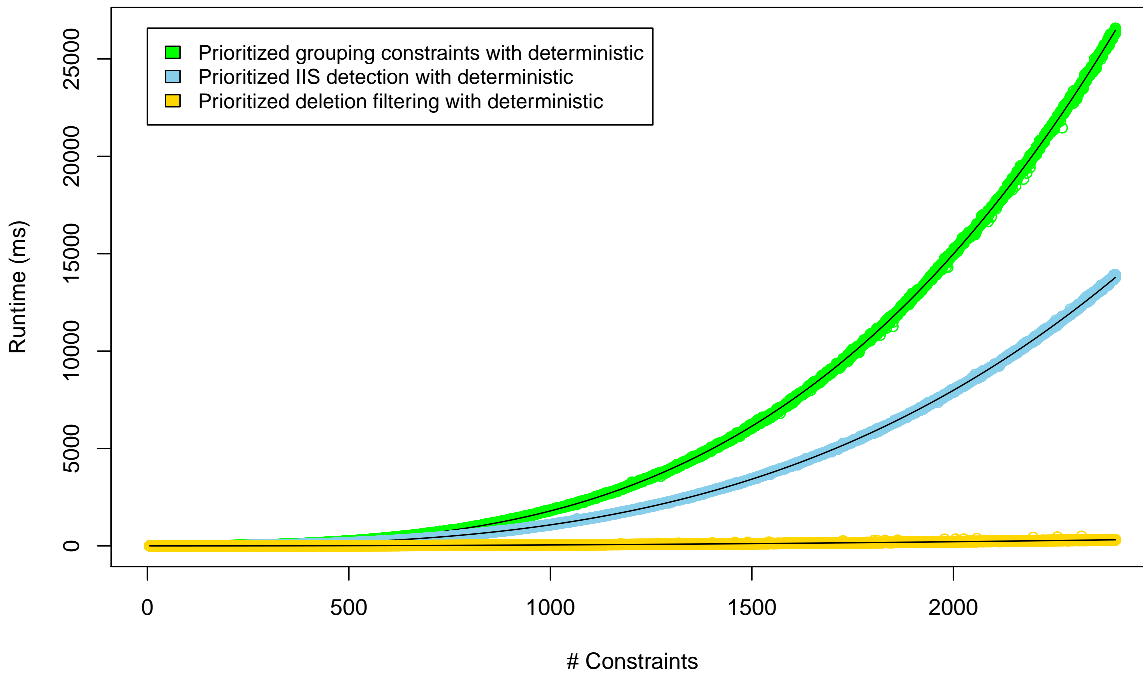


Figure 4.4: Runtime comparison of prioritized IIS detection, prioritized deletion filtering and prioritized grouping constraints with deterministic pivot assignment

ment is more complex and takes longer. The slowest strategy is prioritized grouping constraints with deterministic pivot assignment, followed by prioritized IIS detection with deterministic pivot assignment. Prioritized deletion filtering with deterministic pivot assignment has the best performance. The runtime of prioritized deletion filtering with deterministic pivot assignment appears almost linear in the number of constraints. This is due to the fact that for prioritized deletion filtering, the pivot assignment only needs to be recomputed for each conflicting constraint. The runtime performance of prioritized grouping constraints has the highest volatility. This is due to the fact that the performance of prioritized grouping constraints depends on the distribution of conflicting constraints over the list of constraints. If conflicting constraints are close, the algorithm searches only a small fraction of the whole list. If conflicting constraints are almost equally distributed over the list of constraints, the algorithm searches the whole list.

Figure 4.5 compares all the aforementioned algorithms, except for the slow prioritized IIS detection and prioritized grouping constraints with deterministic pivot assignment, to LINPROG, LP-Solve and QR-decomposition. Generally, all our algorithms perform significantly better than LINPROG, LP-Solve and QR-decomposition, especially for bigger problems, with prioritized grouping constraints with random pivot assignment exhibiting the best runtime behaviour.

Table 4.3 depicts the results of Experiment 3, the comparisons of the algorithms according to

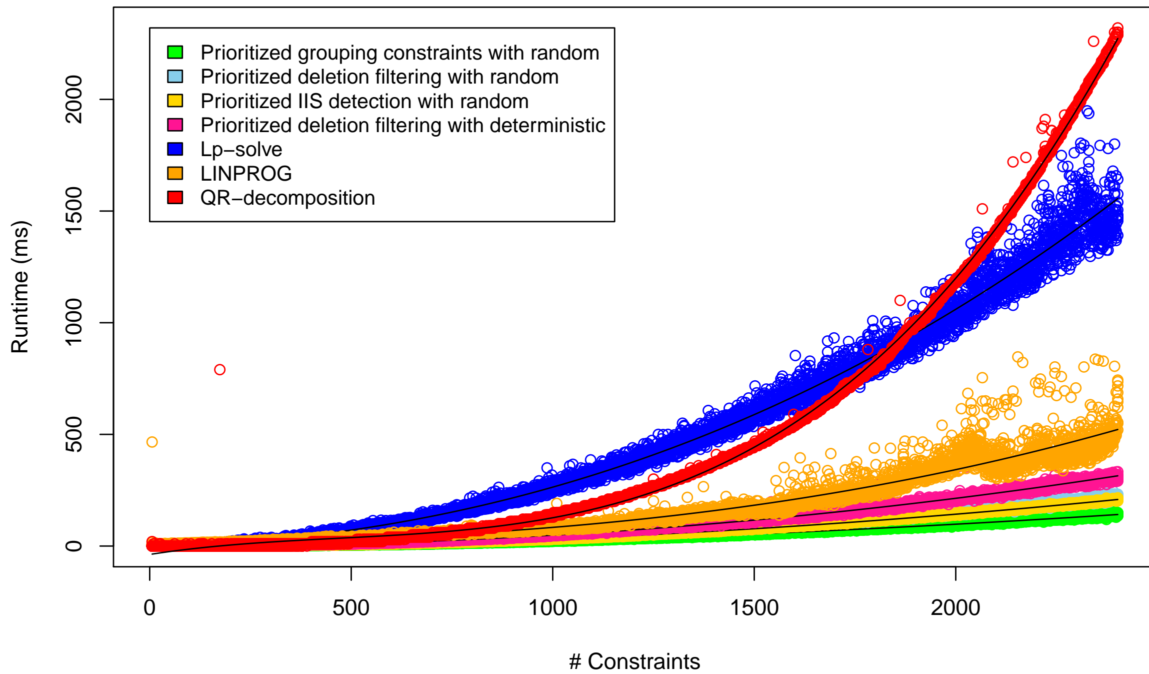


Figure 4.5: Performance comparison of the best solving strategies with LINPROG, LP-Solve and QR-decomposition

Rank	Strategy
1	Pr. IIS detection / random, Pr. grouping constraints / random
2	Pr. deletion filtering / random
3	Pr. grouping constraints / deter., Pr. IIS detection / deter.
4	LINPROG
5	LP-Solve
6	Pr. deletion filtering / deter.

Table 4.3: Comparison of the quality of the solutions produced by the different algorithms (rank 1 = best)

the overall solution quality. The pairwise test results showed that the algorithms can be ordered totally, i.e. each algorithm is better than all the algorithms ranked below it (i.e. with a bigger rank number). The solvers with random pivot assignment produce the best results, with prioritized IIS detection and prioritized grouping constraints at the top. The worst solutions are produced by prioritized deletion filtering with deterministic pivot assignment and the two simplex solvers, LINPROG and LP-Solve. Solvers with a deterministic pivot assignment produce mid-quality solutions.

4.6.3 Discussion

The performance results show that the prioritized grouping constraints algorithm with random pivot assignment is the fastest and that the prioritized IIS detection with deterministic pivot assignment is the slowest. Nevertheless, all proposed algorithms except for prioritized IIS detection and prioritized grouping constraints with deterministic pivot assignment are faster than QR-decomposition, LP-Solve and LINPROG. One likely reason why LINPROG and LP-Solve are slower, is that they are based on the simplex algorithm with one Gauss-Jordan elimination step per iteration, i.e. using a direct method. As described earlier, direct methods suffer from fill-in effects when solving sparse systems, which generally make them inferior to indirect, iterative methods in this case.

Our evaluation has shown that random pivot assignment positively impacts the quality of solutions. One reason may be the common property of randomized algorithms to achieve a good overall behaviour by avoiding systematic worst cases. Prioritized IIS detection and Prioritized grouping constraints yield the best solutions. The reason is that these solvers test constraints step by step. However, the grouping constraints algorithm yields almost as good results as prioritized IIS detection.

It is natural that LINPROG and LP-Solve are outperformed by the newly introduced algorithms in terms of solution quality, since they resolve conflicts differently. If a high-priority constraint needs to be violated, these approaches will not simply "abandon" the constraint, as our algorithms would do. Instead, they still try to minimize this violation, even if this comes at the cost of violating many lower-priority constraints. This means a worse $\bar{\iota}$ value is generated.

If the two properties measured by the above previously described experiments are considered together, the prioritized grouping constraints algorithm with a random pivot assignment is the best. It is the fastest, and its overall solution quality is at the top.

4.7 Summary

The Chapter proposed new algorithms for using SOR for solving constraint-based UI layout problems. In particular, we presented the following contributions:

- Algorithms to resolve conflicts in over-determined specifications by using soft constraints.
- An experimental evaluation that shows that the proposed algorithms find feasible subproblems and outperform a modern linear programming solver, LINPROG, LP-Solve, and a QR-decomposition solvers with respect to execution time.

The work presented in this chapter lays a foundation for the application of iterative methods for solvers of constraint-based UIs. In our evaluation, we identified prioritized grouping constraints

with random pivot assignment as the fastest algorithm, which produces an acceptable quality of solutions.

With the contributions mentioned above, we have demonstrated that indirect methods can efficiently be used for solvers for constraint-based UIs. With the algorithms presented in this chapter, it is possible to bring the benefits of solving sparse matrices efficiently with indirect methods to the domain of constraint-based UI layout.

5

Constraints Reordering Technique for Successive Over-Relaxation

5.1 Introduction

If Successive Over-Relaxation (SOR) is applied to linear systems, the ordering in which constraints are solved affects the convergence behaviour of the algorithm. A bad ordering can slow down the convergence whereas a good ordering can speed it up. To overcome this problem, we propose an algorithm in this chapter that reorders the sequence of constraints used during solving in an optimal way. Our contribution consists of, first, a metric to measure the optimality of a constraint sequence and, second, a simulated annealing based algorithm, which optimizes the order of constraints. This algorithm has to be seen as an addition to our previously presented algorithms for conflict resolution and pivot assignment, which we outlined in Chapter 3 and Chapter 4.

The performance of SOR with sequence optimization is evaluated empirically using randomly generated UI layout specifications of various sizes. The results show that our proposed algorithm outperform Matlab's LINPROG linear optimization package [115], LP-Solve [17] and the implementation of QR-decomposition of the Apache Commons Math Library [8]. LP-Solve is a well-known linear programming solver that has been used for constraint-based UI layout. The implementation of QR-decomposition of the Apache Commons Math Library is an example of a direct method.

The rest of the chapter is organized as follows. Section 5.2 discusses related work about constraint reordering. Section 5.3 puts this research into context with background information about constraint reordering. An algorithm to optimize the solving sequence of constraints is described in Section 5.4. We evaluate the effect of solving sequence optimization in Section 5.5 and draw conclusions in Section 5.6.

5.2 Related Work

Reordering of equation systems is discussed in several domains. Highly related to our contribution are algorithms that reorder square linear systems of equations to reduce the computational effort, and they are explained below.

Some reordering of rows and columns is used to minimize fill-in in direct methods using graph theory and the minimum degree algorithm [1]. The Cuthill-McKee ordering [31] and other related algorithms [107] are also in this category. These reordering techniques try to find a permutation of the unknowns that minimizes the fill-in, changing the sparsity pattern but not the solution.

Benzi et al. [14, 15] tested the Reverse Cuthill-McKee (RCM) and Nested Dissection (ND) reordering algorithms for solving square linear systems and found that reordering of the coefficient matrix reduced the number of iterations when solving with Krylov subspace methods. They also analyzed the convergence behaviour of these reordering algorithms and concluded that some matrices that had not converged before reordering converged afterwards.

Florez et al. [43] present results about the effect of reordering techniques on the rate of convergence of the iterative Krylov subspace methods for non-symmetric sparse linear systems. They used the minimum degree algorithm to minimize fill-in in direct methods similar to Benzi et al. [14, 15]. Their reordering algorithm is useful for direct as well as iterative methods.

Ghidetti et al. [37,50] propose reordering algorithms for solving square linear systems using the generalized minimal residual (GMRES) and conjugate gradient methods. They conclude that reordering algorithms can help in reducing the number of iterations. This can lead to a reduced computational effort for iterative methods.

Hallett and Fisher [69] discuss a constraint reordering approach with SOR for non-linear systems that is based on an extension of the well known convergence theorem that the spectral radius of an iteration matrix of a linear problem has to be less than 1 (see Theorem 2 in Section 2.2.5). They applied a row reordering technique to get maximum pivot coefficients on the diagonal position for square non-linear systems. They formulate on the basis of the convergence theorem an optimization criteria that should approximately hold and present an algorithm to reorder a system in order to optimize the criteria. The algorithm is applied to several econometric models. The authors found that the convergence speed is indeed improved. These results are proved in [68]. Even though the results are promising Hallett and Fisher found that their

approach does not produce optimal reordering in the general case.

All the above mentioned techniques can not be applied directly in our case because we have non-square systems. It was found that reordering the equations can have an effect on the convergence of SOR for solving linear square systems [107]. We propose a more general reordering technique for non-square linear systems. To solve such systems efficiently we define a metric which measures the *goodness of solving a sequence*. Afterwards we describe an optimization procedure which optimizes the sequence according to the defined metric.

5.3 Background: Effects of Constraint Reordering on Convergence of SOR

To specify the context for our algorithm we give a short overview about effects of constraint reordering on the convergence of SOR. Consider the system with the data given in Example 1.

Example 1.
$1x_1 - 0.9x_2 - 0.9x_3 = 0$
$-0.9x_1 + 1x_2 + 0x_3 = 0$
$-0.9x_1 + 0x_2 + 1x_3 = 0$

In this order, this Example does not converge with SOR for any arbitrary initial guess except zero for any relaxation parameter values inside the interval $(0, 2)$.

Proof. The SOR iteration converges for an arbitrary initial guess only if the spectral radius of an iteration matrix is less than one [107]. To show that this does not hold for this example first we derive the SOR iteration matrix and then calculate the spectral radius of the iteration matrix. The SOR iteration matrix is given in (5.1). For any relaxation parameter one of the eigenvalues of this iteration matrix is greater than one. Thus SOR will not converge. □

$$\begin{pmatrix} 1-w & 0.9w & 0.9w \\ 0.9w(1-w) & (0.9w)^2 + (1-w) & (0.9w)^2 \\ 0.9w(1-w) & (0.9w)^2 & (0.9w)^2 + (1-w) \end{pmatrix} \tag{5.1}$$

Consider the system with reordered equations in Example2.

Example 2.
$-0.9x_1 + 1x_2 + 0x_3 = 0$
$1x_1 - 0.9x_2 - 0.9x_3 = 0$
$-0.9x_1 + 0x_2 + 1x_3 = 0$

In this order, the system converges with SOR for any arbitrary initial estimates and relaxation parameter values.

Proof. To check the convergence, we derive the SOR iteration matrix for the reordered Example 2. The reordering used in this example is only a row reordering, so the reordering also changes the pivot assignment. The SOR iteration matrix is given in (5.2).

$$\begin{pmatrix} (0.9)^2(1-w) & 0.9w & 0 \\ -0.9w(1-w) & (-w)^2+(0.9)^2(1-w) & 0 \\ 0 & -(0.9)^2w(1-w) & (1-w) \end{pmatrix} \tag{5.2}$$

All the eigenvalues for this iteration matrix are less than one and thus SOR converges. □

5.4 Optimizing the Solving Sequence of Constraints

The goal of the optimization of the solving sequence of constraints is to have a faster convergence rate, which is based on the observation that the SOR method solves one constraint at a time and uses the updated variable values to solve the next constraint in the list of constraints.

A constraint c_j is used to calculate a value for a variable x_o. This variable has a coefficient a_{jo}, the pivot coefficient of constraint c_j. To calculate the variable value the values of the previous iteration are used. A constraint c_j influences a constraint c_i directly, if its pivot variable appears in c_i (i.e. the variable was changed by SOR using c_j).

To explain this idea we first have to introduce some terms.

Definition 9. A constraint c_j (an influencing constraint of c_i) with pivot coefficient a_{jo} influences another constraint c_i iff

$$a_{io} \neq 0.$$

Definition 10. The influence $\iota_{i,j}$ of an influencing constraint c_j is the fraction of the coefficient of its pivot element in constraint c_i, a_{io}, and the coefficient of the pivot element of constraint c_i, a_{il},[1]

$$\iota_{i,j} = \frac{a_{io}}{a_{il}}$$

The influence of an influencing constraint c_j on an influenced constraint c_i depends on the influencing coefficient a_{io} in c_i and the coefficient a_{il} of the pivot element l of constraint c_i. The greater the influencing coefficient a_{io} compared to a_{il} is, the greater the influence of c_j. If the variable values converge towards the solution of the constraint system, solving one constraint in

[1]This is a common convergence proof for the Gauss-Seidel algorithm [90] applied to square linear systems.

one iteration usually produces an error (ϵ_i) in SOR as each iteration approximates the solution. The error is defined as

$$|\epsilon_i^k| = |b_i - \sum_{j=0}^{j=n} (a_{jo} x_j)|.$$

It is 0 if a possible solution for x is found.

$$\epsilon_i \to 0 \ \forall i.$$

We take equation for the error of one variable value in SOR and generalize it for non-square linear systems. Furthermore, we insert the calculated error values of other variables into the error equation. Thus the error of a constraint c_i in iteration $k+1$ can be expressed dependant on the errors of all influencing constraints and their influence $\iota_{i,j}$

$$|\epsilon_i^{k+1}| = |\sum_{j=1}^{l-1} \iota_{i,j} \epsilon_j^{k+1} + \sum_{j=l+1}^{n} \iota_{i,j} \epsilon_j^k|. \tag{5.3}$$

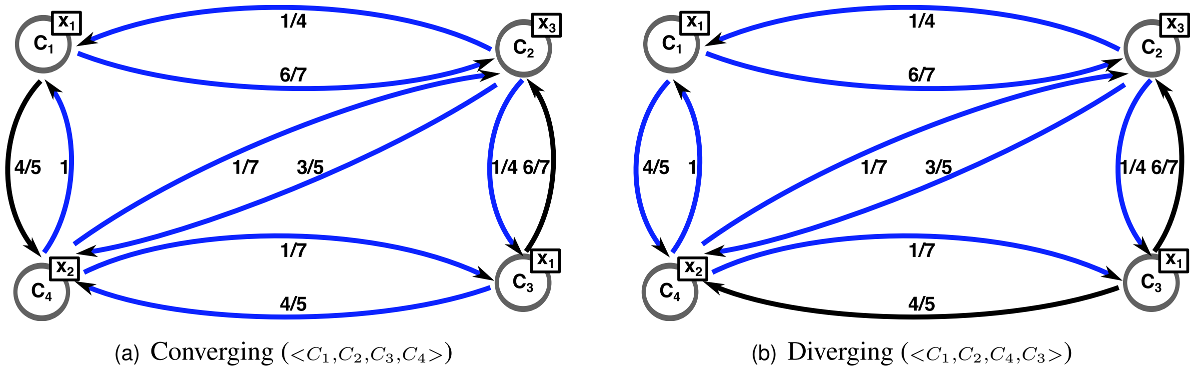
Similar to equation (2.28) when solving equation (5.3) it is clear that the problem converges more quickly if error (ϵ_i)is smaller. We try to reorder the constraints in a way that the error (ϵ_i) becomes smaller and hence contribute towards minimizing the overall error ϵ.

The following example of an over-determined system illustrates the idea of identifying the effect of the constraints on each other:

$$\begin{aligned}
x_1 &= \tfrac{1}{4} + 1x_2 - \tfrac{1}{4}x_3 \quad (C_1) \\
x_3 &= \tfrac{2}{7} + \tfrac{6}{7}x_1 + \tfrac{1}{7}x_2 \quad (C_2) \\
x_1 &= \tfrac{1}{4} + \tfrac{1}{7}x_2 - \tfrac{1}{4}x_3 \quad (C_3) \\
x_2 &= 0 + \tfrac{4}{5}x_1 + \tfrac{3}{5}x_3 \quad (C_4)
\end{aligned}$$

In this example, the sequence, in which the constraints are solved, determines the convergence of the system.

Figure 5.1 identifies the effect of the constraints on each other. The nodes of the figure are the constraints with selected pivot elements, the edges are the influences the constraints have on other constraints. The blue arrows indicate that a direct influencing dependency exists between the two constraints. For example, constraint C_4 influences constraint C_1 by a value of 1, since the pivot variable of C_4 is x_2 and the coefficient of x_2 in C_1 is 1. Constraint C_4 is the constraint whose pivot variable value is calculated before the pivot variable value of C_1 is calculated. A black arrow means there could be an influence if the constraints are reordered. Foe example, C_1 could influence C_4 with pivot coefficient $4/5$ if C_1 is calculated directly before C_4. But since C_3 is calculated before, the variable value of x_1 is overwritten by constraint C_3 and hence there is no direct influence of C_1.



(a) Converging ($<C_1,C_2,C_3,C_4>$) (b) Diverging ($<C_1,C_2,C_4,C_3>$)

Figure 5.1: Graphical representation of problem with 4 constraints

If C_3 is solved prior to C_4, the system converges (see Figure 5.1 a)), otherwise it diverges (see Figure 5.1 b)). Divergence is a rather extreme case of convergence behavior but demonstrates the effect clearly. If $C_3$is used to calculate the value of x_1, which is subsequently used in C_4, the system converges (see Figure 5.1 a)). On the contrary, if C_4 is used to calculate the value of x_2, which is subsequently used in C_3, the system diverges (see Figure 5.1 b)).

Our idea is to find the path that minimizes the influences. Therefore, we, firstly, have to define a metric, which measures the *goodness of solving sequence* and, secondly, we have to describe an optimization procedure, which optimizes the sequence according to the defined metric.

5.4.1 Metric *goodness of solving sequence* (g)

To estimate the convergence behavior, we define a badness value β_i of constraint i and initialize it with 1 ($\beta_i^0 = 1\ \forall i$). The β_i values are a substitution for $epsilon_i$ in (5.3) but initialized with 1. They solely serve the purpose to see how the constraint system behaves after some iterations and how it reacts on changes in the order of constraints. It simplifies the calculation thereby speeding up the calculation. Hence the β_is are calculated in $k = 1..n$ iterations for each constraint according to the equation:

$$\forall C_i : \beta_i^{k+1} = \sum_{j \neq l} \iota_{i,j} \beta_j^k .$$

As for the error ϵ the problem converges if all β_i become less than one.

The β values build a tuple $B =< \beta_1, \ldots, \beta_n >$. Over this tuple, we define the metric *goodness of solving sequence*(g) as the infinity norm over the tuple B.

5.4.2 Algorithm Based on Simulated Annealing

Simulated Annealing is a heuristic inspired by the physical process of annealing, in which as molten metal slowly cools down whereas the molecules optimally arrange themselves in a crystalline framework structure until they reach an optimal solid state [100]. We used Simulated Annealing to optimize the order of constraints according to the metric g, which has to be minimized $g \rightarrow \min$. Algorithm 5.2 transfers Simulated Annealing to the minimization of the g.

1: $s^* \leftarrow s_1 \leftarrow random(S)$
2: $g^* \leftarrow g(s^*)$
3: $n \leftarrow 1$
4: **repeat**
5: $s \leftarrow N(s_n)$
6: **if** $g(s) \geq g(s_n)$ **then**
7: $s_{n+1} \leftarrow s$
8: **end if**
9: **if** $g(s) > g^*$ **then**
10: $g^* \leftarrow g(s)$, $s^* \leftarrow s$
11: **else**
12: $p \leftarrow random([0, 1])$
13: **if** $p \leq p(n)$ **then**
14: $s_{n+1} \leftarrow s$
15: **end if**
16: **end if**
17: n++
18: **until** Stopping condition $\bigotimes$

Figure 5.2: Simulated Annealing algorithm to minimize g with $random()$ randomly selecting elements and $N()$ defining the neighborhood of a given solution (adapted from [100])

A randomly chosen solution candidate (s) from the solution set (S) is randomly altered and tested for its overall performance according to the given objective function, in our case the metric g (line 1). The solution set S consists of all possible permutations of the list of constraints (**C**). A solution candidate s is one permutation that is iteratively compared to a current solution (s_n). If the new solution candidate generates the same or a lower g, it is the new solution for the next iteration (s_{n+1}). However, if only strictly better solutions are taken, it is likely that the procedure will get stuck in local optima and never find a global optimum. Thus, Simulated Annealing sometimes accepts worse solutions. The probability (p) of accepting such a negative change depends on the amount the g increases and the number of processed iterations. The longer the procedure runs the more unlikely it is that the procedure accepts a worse solution [100].

To decrease the acceptance rate, the acceptance probability ($p()$) follows an exponential

probability distribution

$$p(n) = e^{-\frac{\Delta\Pi_n}{T(t)}},$$

whereas $T(t)$ is the *temperature* at step t and $\Delta\Pi_n = \Pi(s) - \Pi(s_n)$ the difference between the current solution and the temporary best solution. The temperature $T(t)$ is a stepwise decreasing function [100].

The procedure stops ($\bigotimes$) after N_{stop} iterations or after N_{change} iterations without improvements in g by ϵ (line 18).

5.5 Experimental Evaluation

We conducted two different experiments to evaluate (i) the convergence behaviour and (ii) the performance in terms of computation time. The experiments were conducted as follows.

5.5.1 Methodology

We implemented the problem and solution procedure by using the *Opt4J* tool suite [85]. This is a framework for meta-heuristics and comprises an optimization runner, a configuration interface, and many conveniences for testing implemented solution procedures.

In our experiments we used the following setup: a desktop computer with Intel Core 2 Duo 3GHz processor under Windows 7 running an Oracle Java virtual machine.

UI-Layout specifications were randomly generated using the test data generator described in Chapter 4. For each experiment the same set of test data was used. The specification size was varied from 4 to 2402 constraints in increments of 4 (2 new constraints for the position and 2 new constraints for the preferred size of a new widget). For each size 10 different layouts were generated resulting in a total of 6000 different layout specifications which were evaluated. A relaxation parameter (ω) of 0.7 and a tolerance of 0.01 were used for SOR.

In experiment (i) we investigated the convergence behaviour of each algorithm by measuring the number of sub-optimal solutions. A solution is sub-optimal if the error of a constraint (the difference between the right hand and left hand sides) is greater than the tolerance.

In experiment (ii) we measured the performance in terms of computational time T in milliseconds (ms), depending on the problem size (number of constraints c). We solved each problem one time with sequence optimization and the other time without sequence optimization. For the solver with sequence optimization we counted the reordering of the constraints to the measured solving time.

As a reference, all the generated specifications were also solved with Matlab's LINPROG solver [115], LP-Solve [17] and the QR-decomposition implementation in the *Apache Commons Mathematics Library* [8]. We selected LINPROG because it is a widely known for its

speed [2] and LP-Solve because it has been previously used to solve constraint-based UI layout problems [86]. We included QR-decomposition implementation to have an implementation of a direct method as a reference as well.

For each run the solving time and the optimality status was taken.

5.5.2 Results

The analysis of results is the same as in previous Chapters 4 and 6. In experiment (i) we investigated the convergence behaviour of our algorithm. We found that the algorithm converges with and without sequence optimization in the end. This result is obvious since the conflict resolution algorithm is designed to find a solvable subproblem and shows that the conflict resolution algorithm works.

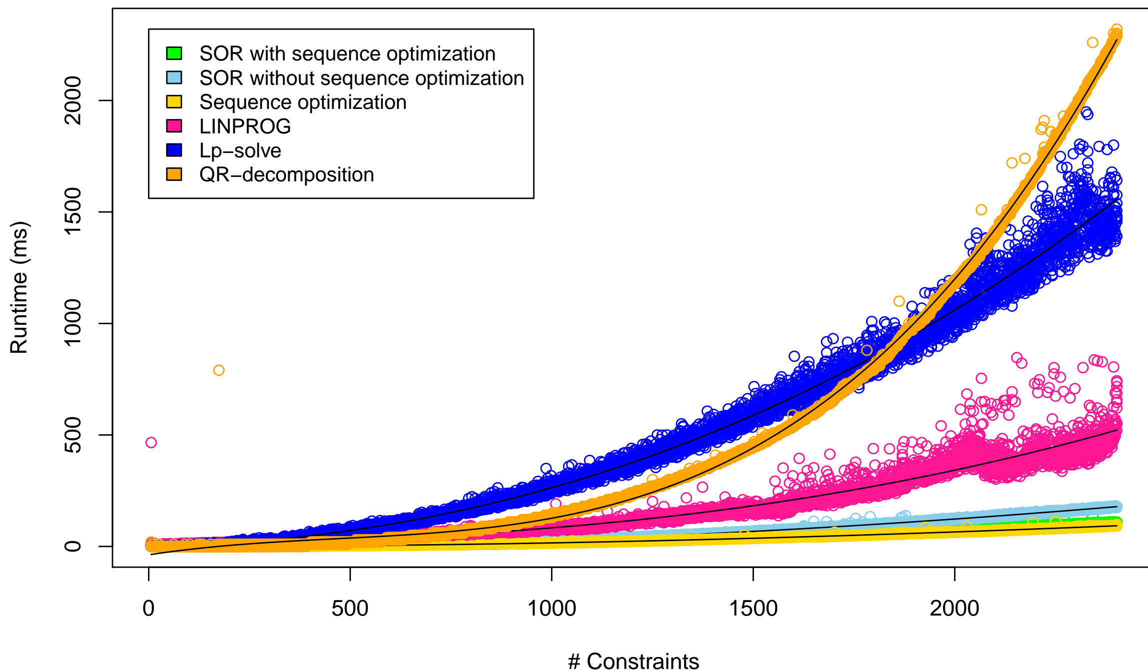


Figure 5.3: Runtime comparison of LINPROG, LP-Solve and QR-decomposition with our new solvers

In experiment (ii) we investigated the computational time behaviour with and without sequence optimization. To establish the trend of the performance of the algorithms, we defined some regression models (linear, quadratic, log, cubic). We found that the best fitting model is the polynomial model

$$T = \beta_0 + \beta_1 c + \beta_2 c^2 + \beta_3 c^3 + \epsilon$$

[2]http://plato.asu.edu/ftp/lpfree.html

which gave us a good fit for the performance data. Key parameters of the models are depicted in Table 5.2; a graphical representation of the models can be found in Figures 5.4 and Figures 5.3. Table 5.1 explains the symbols used.

Table 5.1: Symbols used for the performance regression model

Symbol	Explanation
β_0	Intercept of the regression model
β_{1-3}	Estimated model parameters
c	Number of constraints
T	Measured time in milliseconds
R^2	Coefficient of determination of the regression models

Figure 5.4 illustrates the performance comparison of sequence optimization, for SOR with sequence optimization and without sequence optimization using a random pivot assignment and the prioritized grouping constraints algorithms. Both algorithms with sequence optimization exhibited a better performance than the one without sequence optimization. However, it also introduced a slightly higher variance in the data.

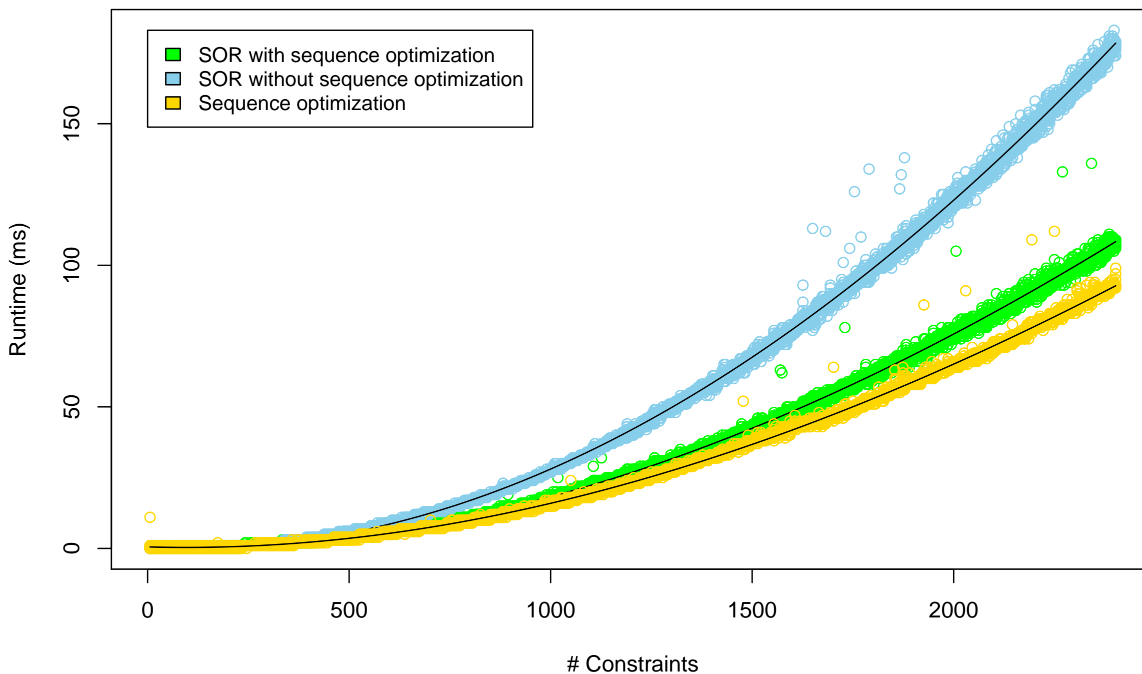


Figure 5.4: Runtime with and without sequence optimization

Figure 5.3 illustrates the performance of solvers based on LINPROG, LP-Solve and QR-decomposition compared to our new algorithms. Generally, both sequence optimization algo-

Table 5.2: Regression models for the different solving strategies

Strategy	β_0	β_1	β_2	β_3	R^2
SOR with so	0.7793^{***}	$-5.161 \cdot 10^{-03***}$	$2.389 \cdot 10^{-05***}$	$-1.290 \cdot 10^{-09***}$	0.9983
SOR w/o so	0.8770^{***}	$-9.392 \cdot 10^{-03***}$	$3.787 \cdot 10^{-05***}$	$-1.324 \cdot 10^{-09***}$	0.999
With so	0.5345^{***}	$-3.860 \cdot 10^{-03***}$	$2.045 \cdot 10^{-05***}$	$-1.118 \cdot 10^{-09***}$	0.9984
LINPROG	18.29^{***}	$1.591 \cdot 10^{-04}$	$4.934 \cdot 10^{-05***}$	$1.577 \cdot 10^{-08***}$	0.9367
LP-Solve	-2.491^{***}	$3.924 \cdot 10^{-02***}$	$2.079 \cdot 10^{-04***}$	$1.904 \cdot 10^{-08***}$	0.9900
QR-Decomposition	-37.70^{***}	0.2802^{***}	$-4.009 \cdot 10^{-04***}$	$2.850 \cdot 10^{-07***}$	0.9989

Significance codes: *** $p < 0.001$, so: sequence optimization

rithms perform significantly better than the references LINPROG, LP-Solve and QR-decomposition and the results confirmed previous observations [71].

5.5.3 Discussion

The results of the experiment show that sequence ordering according to the metric g reduces the solving time significantly. However, they also show that sequence optimization exhibits a slightly greater variance in the solving time. There could be two reasons for this: first, the runtime behaviour of the ordering step with Simulated Annealing is less predictable due to the random element of the algorithm and, second, is it possible that there are more influences on the optimality of a sequence than are measured with g.

Generally, sequence optimization improves the convergence speed and should therefore be included in implementations of solvers for constraint-based UI solvers based on SOR.

5.6 Summary

In this chapter we presented a Simulated Annealing based algorithm to optimize the solving sequence of constraints to speed up the convergence of the SOR method for over-determined systems in constraint-based UI problems. The evaluation has shown that the reordering of the sequence has an impact on the convergence behaviour and that our metric g reliably measures the adequacy of a solving sequence of a list of constraints. The evaluation with synthetically generated test problems from the domain of constraint-based UIs shows that the convergence speed is improved. Thus, a preceding sequence optimization step can improve the runtime of SOR-based UI-solvers.

6
Constraint-wise Under-relaxation

6.1 Introduction

When using the Successive Over-Relaxation (SOR) method for solving linear constraints for GUI layout problems, one needs an estimate of the relaxation parameter to make it converge. This can be achieved by estimating the eigenvalues of an iteration matrix of the SOR method [123]. For small systems of equations, good estimates for the optimal relaxation parameter may be possible because it is easy to estimate the eigenvalues of the iteration matrix; for large systems however this may not be the case [123].

The speed of convergence of basic iterative methods depends on the spectral radius of an iteration matrix, defined as the absolute value of the maximum eigenvalue of an iteration matrix (see Definition 3 for an iteration matrix in Chapter 2).

In this chapter, one of our significant approach in increasing the convergence rate of SOR is the Constraint-wise Under-relaxation (CWU). The main difference between CWU approach and other approaches such as [126] is that our approach chooses different relaxation parameters for each constraint while other approaches choose one global relaxation parameter for all constraints.

For each constraint, the under-relaxation parameter is chosen in a way that a pivot coefficient, which is not row dominant, becomes a weight as if it would be row dominant. A coefficient is row dominant if it is greater than the absolute value of the sum of all other coefficients of a row.

The relaxation parameter (ω) for SOR method lies inside the interval $(0, 2)$. $\omega > 1$ indicates over-relaxation and $\omega < 1$ indicates under-relaxation. Over-relaxation helps in speeding up convergence of SOR, but we cannot use over-relaxation because it diverges for GUI layout problems. When we use SOR method, a best relaxation parameter value for GUIs is 0.7 because it converges for GUI layout problems that has been shown empirically in Chapter 4, Chapter 5 and Chapter 8. In this chapter we use a relaxation parameter based on CWU formula that is given as follows.

Definition 11. CWU formula: The optimal choice of (ω) for SOR, denoted by w_i is given by $w_i = \frac{1}{\|a_{i1} \ldots a_{i(i-1)}, 0, \ldots a_{in}\|}$.

We know that if a matrix is strictly diagonally dominant then the convergence of SOR is guaranteed. If we choose a relaxation parameter based on CWU formula then it can scale the coefficient matrix in a way that it is equal to diagonally dominant criteria (see definition 2.19 in Chapter 2).

We were unable to provide theoretical results on the convergence of SOR but we show empirically that we can increase the computational performance of an SOR-based constraint solver if we choose CWU. In this chapter, we report on experiments to test this approach experimentally for solving linear constraint problems for GUI layout. In our experiments, we measured the convergence and solving time for randomly generated GUI layout specifications of various sizes.

The remainder of the chapter is structured as follows. We describe a related work in Section 6.2. The methodology used as well as the results of the evaluation are given in Section 6.3. Section 6.4 wraps the chapter up with conclusions.

6.2 Related Work

For a general system $Ax = b$, there is no formula for finding the optimal relaxation parameter. However, Young [126] proposed a formula for calculating an optimal relaxation parameter for consistently ordered and 2-cyclic matrices which is not useful for GUI layout problems because this formula applies only on specific class of matrices. Furthermore, Young [126] proved that if the spectral radius of an iteration matrix is strictly less than one then SOR convergence can be maximized and guaranteed.

Dancis [32] proposed that polynomial acceleration can improve the convergence of SOR if the optimal relaxation parameter is selected. Subsequently, Eiermann et al. [39] criticized Dancis theory and showed that polynomial acceleration of SOR is not faster than SOR with the optimal relaxation parameter proposed by Young [126]. There are some other approaches being developed for selection of relaxation parameters [106, 113] which use different relaxation

parameters at each point. One of the drawbacks of these methods is that they are designed for matrices with certain characteristic properties and therefore are not generally applicable.

6.3 Experimental Evaluation

In this section, we describe an experimental evaluation of the proposed algorithms. Two different experiments were conducted to evaluate (1) the convergence behaviour and (2) the performance in terms of computation time. The experiments were arranged as follows.

6.3.1 Methodology

The methodology is the same as in previous Chapters 4 and 5. For experiments we used the same computer and test data generator, but instrumentalized the algorithms differently. We used the following setup: a desktop computer with Intel i5 3.3GHz processor under 64-bit Windows 7, running an Oracle Java virtual machine. Layout specifications were randomly generated using the test data generator described in Chapter 4. For each experiment the same set of test data was used. The specification size was varied from 4 to 2402 constraints, in increments of 4 (2 new constraints for the position and 2 new constraint for the preferred size of a new widget). For each size, 10 different layouts were generated, resulting in a total of 6000 different layout specifications that were evaluated. A tolerance of 0.01 was used for SOR.

In experiment 1, we investigated the convergence behaviour of each algorithm by measuring the number of sub-optimal solutions. A solution is sub-optimal if the error of a constraint (the difference between right-hand and left-hand sides) is greater than the tolerance.

In experiment 2, we measured the performance in terms of computational time T in milliseconds (ms), depending on the problem size measured in number of constraints c. Each of the proposed algorithms was used to solve each of the problems of the test data set, and the time taken was recorded. As a reference, all the generated specifications were also solved with Matlab's LINPROG solver [115] and LP-Solve [17]. We selected these two solvers because LINPROG is widely known for its speed [1], and LP-Solve has been previously used to solve constraint-based UI layout problems [86]. Additionally, we wanted to know how well our algorithms could compete with a direct method. Hence we also used the implementation of QR-decomposition in the Apache Commons Mathematics Library [8], which is a freely available open-source library.

6.3.2 Results

The analysis of results is the same as in previous Chapters 4 and 5. In experiment 1, we investigated the convergence behaviour of our approach. We found that the algorithm converges

[1] http://plato.asu.edu/ftp/lpfree.html

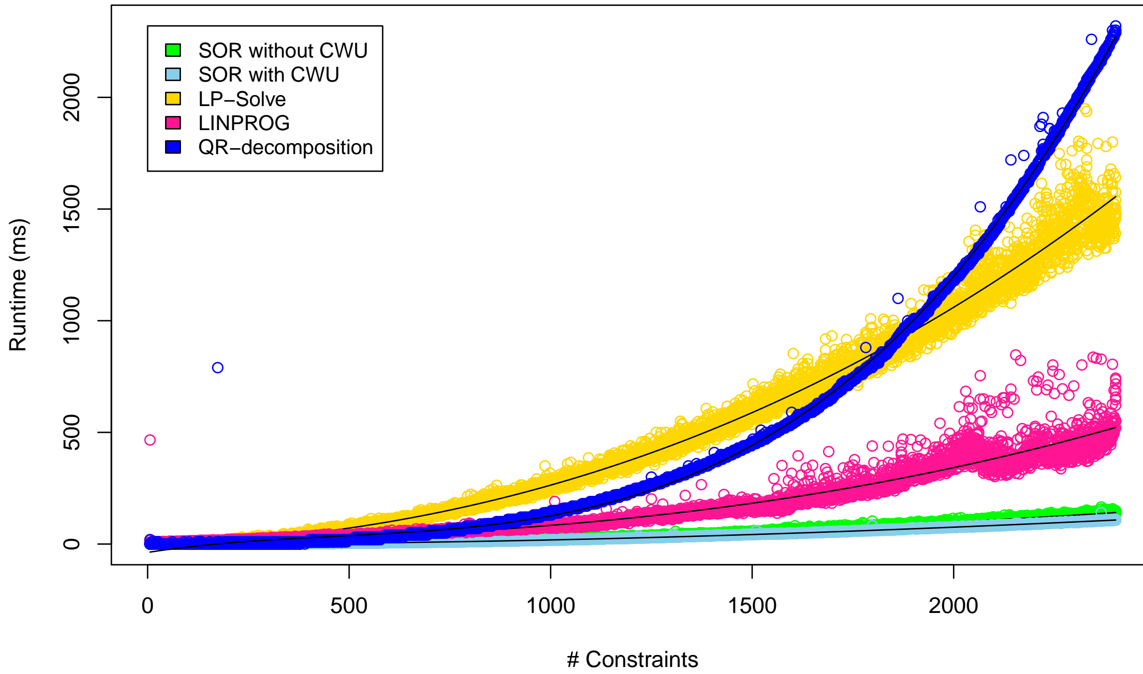


Figure 6.1: Performance comparison of the best solving strategies with LINPROG, LP-Solve and QR-decomposition

with and without CWU. This result is obvious since the conflict resolution algorithm is designed to find a solvable subproblem and shows that the conflict resolution algorithm works.

In experiment 2, we used different regression models (linear, quadratic, log and cubic) to analyze the trends of the computational performance of the algorithms. We found that the best-fitting model is the polynomial model

$$T = \beta_0 + \beta_1 c + \beta_2 c^2 + \beta_3 c^3 + \epsilon.$$

Key parameters of the models are depicted in Table 6.2. Table 6.1 explains the symbols used.

Table 6.1: Symbols used for the performance regression model

Symbol	Explanation
β_0	Intercept of the regression model
β_{1-3}	Estimated model parameters
c	Number of constraints
T	Measured time in milliseconds
R^2	Coefficient of determination of the estimated regression models

Table 6.2: Regression models for the different solving strategies

Strategy	β_0	β_1	β_2	β_3	R^2
SOR without CWU	1.357^{***}	$-1.241 \cdot 10^{-02***}$	$5.035 \cdot 10^{-05***}$	$-2.035 \cdot 10^{-09***}$	0.9997
SOR with CWU	1.035^{***}	$-1.112 \cdot 10^{-02***}$	$4.278 \cdot 10^{-05***}$	$-9.176 \cdot 10^{-10***}$	0.9994
LINPROG	18.29^{***}	$1.591 \cdot 10^{-04}$	$4.934 \cdot 10^{-05***}$	$1.577 \cdot 10^{-08***}$	0.9367
LP-Solve	-2.491^{***}	$3.924 \cdot 10^{-02***}$	$2.079 \cdot 10^{-04***}$	$1.904 \cdot 10^{-08***}$	0.9900
QR-Decomposition	-37.70^{***}	0.2802^{***}	$-4.009 \cdot 10^{-04***}$	$2.850 \cdot 10^{-07***}$	0.9989

Significance codes: *** $p < 0.001$, , CWU: Constraint-wise Under-relaxation

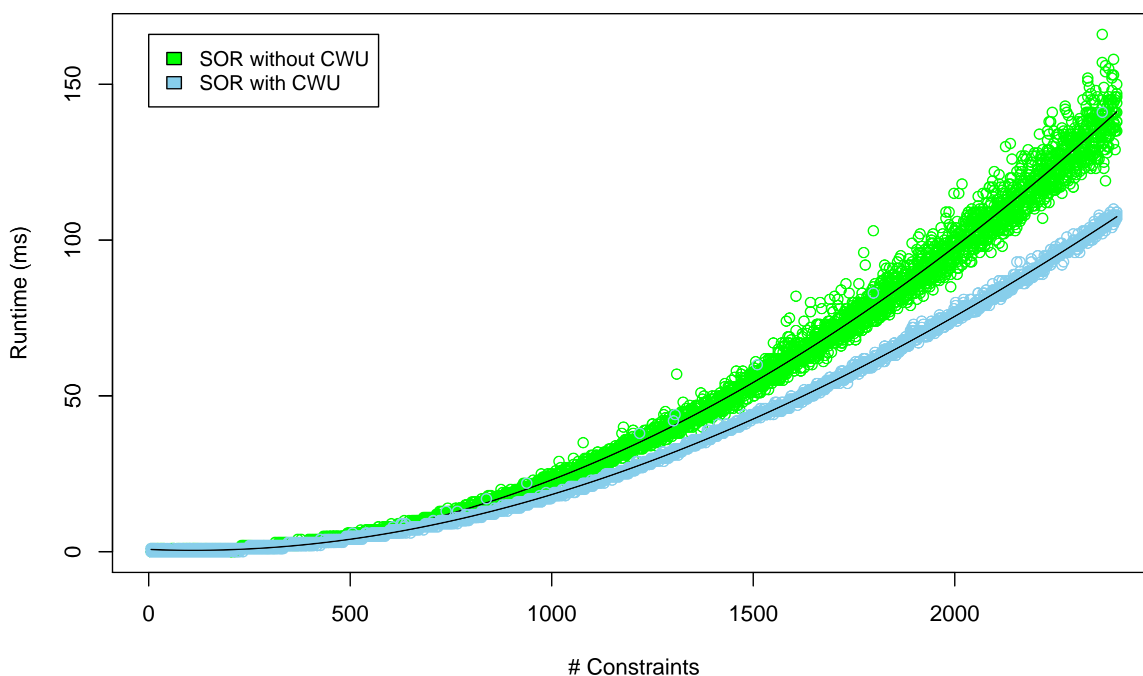


Figure 6.2: Performance comparison of SOR with and without CWU

For some strategies, some parameters do not have a significant effect. This can be interpreted as the complexity of the algorithm not following a certain polynomial trend.

Figure 6.2 illustrates the performance comparison of SOR with and without CWU. As the graphs indicate, SOR with CWU exhibits better performance than SOR without CWU. Figure 6.1 compares our proposed algorithms to LINPROG, LP-Solve and QR-decomposition. Generally, our SOR algorithm with CWU performs significantly better than LINPROG, LP-Solve and QR-decomposition.

6.3.3 Discussion

The performance results show that the SOR with CWU is the fastest and QR-decomposition is the slowest. All our proposed algorithms are faster than QR-decomposition, LP-Solve and LINPROG. As discussed earlier one plausible reason why LINPROG and LP-Solve are slower, is that they are based on the simplex algorithm with one Gauss Jordan elimination step per iteration, i.e. they use a direct method. As described earlier, direct methods suffer from fill-in effects when solving sparse systems, which generally makes them inferior to indirect, iterative methods in this case.

One reason why adjusting one fixed relaxation parameter for all constraints is slower is that it takes more time to decrease the norm of the error because for some constraints this fixed relaxation parameter can increase the error.

6.4 Summary

This chapter evaluated the use of CWU to improve the efficiency of our proposed algorithms for solving constraint-based UI layout problems. We identified SOR with the CWU as the fastest algorithm.

7 Kaczmarz Algorithm with Soft Constraints

7.1 Introduction

The Kaczmarz method [78] is an iterative method for solving large systems of equations that projects iterates orthogonally onto the solution space of each equation. In contrast to direct methods such as Gaussian elimination or QR-factorization, this algorithm is efficient for problems with sparse matrices, as they appear in constraint-based user interface (UI) layout specifications.

Starting with an initial guess, the Kaczmarz algorithm selects a row index of the matrix and projects the current iterate onto the solution space of that equation, refining the solution until a sufficient precision is reached. Because it does not need any pivot assignment, it is ideal for highly over-determined linear systems, as in many linear problems including the constraint-based UI layout. In our previous Chapters 3 and 4, we proposed extensions to the original SOR method to deal with this issue, however, the Kaczmarz method is inherently better suited to solve non-square matrices.

Despite its efficiency for sparse systems, the Kaczmarz method is currently not used for constraint-based UI layout. The reasons for this are two-fold. First, constraint-based UI layout contains linear equality and *inequality* constraints for specifying relationships among objects such as “inside”, “above”, “below”, “left-of”, “right-of” and “overlap”. Although the Kaczmarz

algorithm and its variants are not designed to handle inequality constraints, preliminary work on the Kaczmarz method for inequality constraints suggests the natural adaptation which ignores inequality constraints if they are already satisfied, and otherwise treats them as equalities [82]. We also adapt this heuristic approach for the UI problem.

The second issue the UI problem faces, like many other problems, is that the system may contain *conflicting* constraints. This may happen by over-constraining, i.e. by adding too many constraints, making the system infeasible. If a specification contains conflicting constraints, the common Kaczmarz method simply will not converge. To resolve conflicts, *soft constraints* can be introduced. In contrast to the usual *hard* constraints, which cannot be violated, soft constraints may be violated as much as necessary if no other solution can be found. Soft constraints can be prioritized so that in a conflict between two soft constraints only the soft constraint with the lower priority is violated. This leads naturally to the notion of *constraint hierarchies*, where all constraints are essentially soft constraints, and the constraints that are considered "hard" simply have the highest priorities [21]. Using only soft constraints has the advantage that a problem is always solvable, which cannot be guaranteed if only hard constraints are used.

In this chapter we use two conflict resolution algorithms for solving systems of prioritized linear constraints with the Kaczmarz method. In the first algorithm non-conflicting constraints are successively added in descending order of priority. In the second algorithm, constraints are added or removed and the binary search algorithm is adapted to the problem of searching for the best conflict-free subproblem. These algorithms yield conflict-free sub-problems to a given problem. These algorithms are explained in detail in Chapter 4. Algorithms for finding feasible subsystems already exists, but they differ from our approaches as they do not take into account prioritized constraints [29].

We also use different techniques for optimizing the Kaczmarz algorithm similar to our procedure with SOR, these techniques are described in detail in Chapter 6 and Chapter 8. Furthermore, in this chapter we use Least Squares Kaczmarz with a cooling function for solving constraints for attractive GUIs.

With the presented conflict resolution algorithms Kaczmarz can be applied to linear constraint problems, for example in the domain of constraint-based UI layouts. They were experimentally evaluated with regard to convergence and performance, using randomly generated UI layout specifications. The results show that most of the proposed algorithms are optimal and efficient. Furthermore, we observe that our implemented solvers outperform Matlab's LINPROG linear optimization package [115], LP-Solve [17] and the implementation of QR-decomposition of the Apache Commons Math Library [8]. LP-Solve is a well-known linear programming solver that has been used for constraint-based UI layout. The implementation of QR-decomposition of the Apache Commons Math Library is an example of a direct method.

The remainder of the chapter is organized as follows. We discuss a related work in Section 7.2. In Section 7.3 we describe the Kaczmarz method in detail, and introduce our methods

for soft constraints in Chapter 4. An optimization of Kaczmarz algorithm is described in Section 7.4. We introduce an approach about constraint solving for attractive GUIs in Section 7.5. The methodology as well as the results of the evaluation are shown in Section 7.6. Section 7.7 wraps the chapter up with conclusions and an outlook to future work.

7.2 Related Work

Different direct and iterative methods exist which can solve least squares problems. If linear systems of equations are consistent then the Kaczmarz algorithm converges towards a least squares solution. If the system is inconsistent then every sub-sequence of cycles through the system, converges, but not necessarily to a least squares solution [38]. There are several techniques that have been proposed to deal with these inconsistencies. Herman [38] proposed that the Kaczmarz algorithm converges towards weighted least squares solution if iterating both in x and dual variable r. Popa analyzed a similar approach for solving least squares problems [101,102]. Censor showed that if the relaxation parameter goes to zero then the Kaczmarz method converges towards a weighted least squares solution for inconsistent systems [25]. If the system is unsolvable then the Kaczmarz algorithm converges if relaxation parameter is kept fixed at certain intervals [58].

QOCA [22] uses the active set algorithm for solving quadratic programming problem for graphical user interface layout. The latest work [127] on constraint based GUIs uses a quadratic solving strategy which they find better than linear solving strategies. They [127] implemented the active set method for solving a quadratic objective function subject to some linear constraints.

Most of the research related to GUI layout involves various algorithms for solving constraint hierarchies. Research related to constraint-based UI layout has provided results in the form of tools [65,66] and algorithms [11,22] for specific tasks. Our work is concerned with two different aspects. We must find a solution for linear inequality constraints with iterative methods, while also handling soft constraints. We discuss related work for both aspects in turn below.

Various algorithms are proposed for solving linear inequality constraints in the UI layout. The Indigo algorithm [20] uses interval propagation to solve acyclic collections of inequality constraints, however, it does not handle simultaneous equality and inequality constraints. This is overcome by the Detail method [64,67], which can solve linear equality and inequality constraints simultaneously and uses Gaussian-elimination to solve constraints simultaneously. The Cassowary solver [11] can also handle linear inequalities. It uses the simplex algorithm, and inequalities are solved by introducing slack variables. The simplex algorithm is an iterative method but requires one Gaussian elimination step per iteration which makes this method slower than most other iterative methods. QOCA [91] intends to overcome the difficulties in maximizing the efficiency and facilitating the re-use of the solver in other applications. This

solver introduces slack variables to convert inequality constraints into equality constraints in a similar way to the Cassowary solver. The HiRise constraint solver [65] resolves both equality and inequality constraints in combination with quasi-linear optimization.

However, none of the algorithms discussed above apply Kaczmarz for linear and least squares solutions for UI layout. Because the constraint-based UI layout problem contains inequality and soft constraints, existing UI layout solvers use algorithms other than Kaczmarz. Some of these solvers are already discussed in Chapter 4.

7.3 Kaczmarz Method

The Kaczmarz method is an iterative method used for solving large-scale over-determined linear systems of equations [78]. It is also used in tomography, and in that setting is called the "algebraic reconstruction technique" (ART) [55]. Given a system of *m* equations and *n* variables of the form

$$Ax = b, \tag{7.1}$$

the Kaczmarz method projects orthogonally onto the solution hyperplane of each constraint in the system sequentially. The algorithm can thus be described as follows.

$$x_{k+1} = x_k + \omega \frac{(b_i - a_i \cdot x_k)a_i}{\|a_i\|^2} \tag{7.2}$$

where x_k is the k-th iterate, $i = (k \mod m) + 1$ (for deterministic Kaczmarz), a_i is the i-th row of the matrix A, b_i is the i-th component of the right-hand side vector, and $\|a\|$ denotes the Euclidean norm of the vector a. Alternatively, to randomize the Kaczmarz method, we can choose a random i with $1 \leq i \leq m$ for each k.

Starting with an initial estimate x_0, the method projects the current iterate onto the solution space of the next equation as shown in Figure 7.1. The algorithm iterates until the relative approximate error is less than a pre-specified tolerance. ω is an optional relaxation parameter that is set to 1 in the original Kaczmarz method. The runtime complexity for the Kaczmarz method is $O(n)$.

7.3.1 Convergence

The Kaczmarz method is guaranteed to converge if ω lies inside the interval $(0, 2)$ [78,95]. The convergence bevaviour of Kaczmarz is shown in Figure 7.2.

In this section we give the convergence proof using the terminology explained below.

Lemma 2 (Translation invariance)**.** Let the Kaczmarz method for $Ax = b$ converge to $\bar{x}$ starting with x_0. Then the Kaczmarz method for the homogeneous system $Ay = 0$ starting with $y_0 = x_0 - \bar{x}$ will have the same convergence behaviour, i.e. $y_k = x_k - \bar{x}$ for all k.

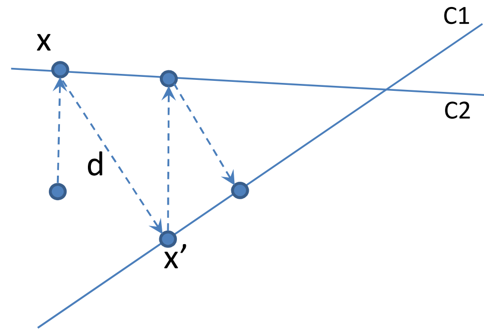


Figure 7.1: Kaczmarz method overview

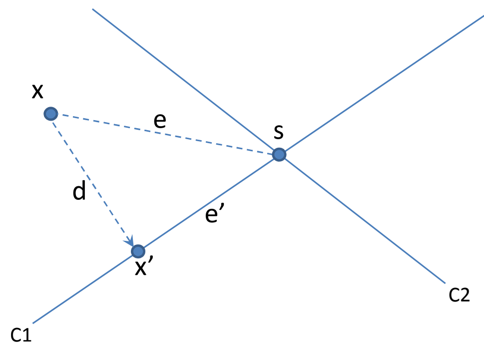


Figure 7.2: Kaczmarz method convergence

The proof is by induction. The induction step follows trivially from the linear definition of the iteration step.

Lemma 3 (Convergence of homogeneous system)**.** The Kaczmarz method for the homogeneous system $Ay = 0$ with nonsingular A converges exponentially for every initial guess y_0.

Proof. The Kaczmarz method is a linear method; by definition, the change to the estimate in every iteration step is a linear function that can be modelled with an iteration matrix $K_i(A)$. We demonstrate that the spectral radius ρ fulfills $\rho(K_i(A)) = 1$. It suffices to show that all vectors are transformed to vectors of shorter or equal size (if $\rho(K_i(A)) > 1$ there would be a vector that gets longer). We observe that each solution hyperplane of the homogeneous system goes through the origin. Since the iteration step performs an orthogonal projection, for $\omega = 1$ the origin and the points y_k and y_{k+1} form a triangle with a right angle at y_{k+1}. Hence

$$\|y_{k+1}\| \leq \|y_k\| \tag{7.3}$$

by Pythagoras. For $0 < \omega < 2$, y_{k+1} is equal in length to a weighted vector sum of y_k and the result for $\omega = 1$, so its length is intermediate and equation ((7.3)) still holds. Hence we have shown that $\rho(K_i(A)) \leq 1$. But if the estimate is already a solution for the constraint, then it

does not move, hence giving $\rho(K_i(A)) = 1$. We now look at the product

$$K(A) = \prod_{1 \le i \le m} K_i(A). \tag{7.4}$$

For any y_0 we have

$$K(A)y_0 = y_m. \tag{7.5}$$

We now show for any $y_0 \neq 0$ that $\|y_0\| > \|y_m\|$. There must be one i so that $y_{i-1} \neq y_i$, since A is nonsingular and hence $y_0 \neq 0$ cannot fulfill all constraints at once. Since in step i we have $y_{i-1} \neq y_i$, we also have by Pythagoras $\|y_{i-1}\| > \|y_i\|$, as explained above. In all other steps the error does not increase, hence we know

$$\|K(A)z\| < \|z\|. \tag{7.6}$$

This means, overall we know $\rho(K(A)) = c < 1$. Hence the error of the estimate decreases over the course of m iterations by at least c, and overall we get an exponential convergence behaviour with base $\leq \sqrt[m]{c}$. □

The proof can be easily generalized to a singular A by only considering the nonsingular orthogonal subspace. Both lemmata together clearly give:

Theorem 4. The Kaczmarz method for $Ax = b$ converges for every initial guess x_0.

The constant c is a characteristic of the problem matrix A, similar to the condition number. Since the convergence rate of the Kaczmarz method depends on c, it is imaginable that preconditioners could be used to reduce c and enhance the convergence speed.

As described by (7.2), the convergence rate of the Kaczmarz method may depend on the ordering of the rows of the matrix A. A problematic ordering can lead to a drastically reduced rate of convergence. To overcome this, a randomized variant can be used. Strohmer and Vershynin proposed a further variant with weighted probabilities proportional to the norm of the ith row [114].

In the inconsistent case, it has been shown that the method exhibits the same convergence down to a threshold [97], and modified methods even converge to the least-squares solution [26, 49, 129]. The convergence rate can further be improved by selecting blocks of rows at a time for the projection [38, 40, 96, 98].

7.3.2 Inequalities

The Kaczmarz method supports only linear equations, but we extend this algorithm for solving linear inequalities in a natural way, as in [82]. In each iteration, the algorithm ignores inequal-

ities if they are satisfied, and otherwise treats them as if they were equations. This means that inequalities influence the solving process only if this is necessary.

7.4 Optimization of Kaczmarz

We apply different techniques for optimizing the Kaczmarz algorithm similar to our procedure with SOR. Some of these techniques accelerate the convergence of the Kaczmarz algorithm for solving GUI layout problems and some do not effect the convergence speed at all. Techniques used to enhance the convergence speed of SOR include constraint reordering, Constraint-wise Under-relaxation (CWU) and a warm-start strategy.

Constraint reordering is not applicable to the Kaczmarz method. With SOR we used the constraint reordering algorithm to first reorder the sequence of constraints based on pivot assignment and then use a simulated annealing (SA) based algorithm to optimize the order of constraints. The Kaczmarz algorithm is designed to solve non-square systems so we do not need pivot assignments. Our constraint reordering algorithm for SOR is based on pivot assignment therefore can not be immediately applied to the Kaczmarz method. Constraint reordering is explained in detail in Chapter 5.

CWU is applicable to the Kaczmarz algorithm but its efficiency is not much improved by using this technique. CWU for SOR is described in detail in Chapter 6. Kaczmarz with CWU gives similar results as Kaczmarz with over-relaxation as shown in Figure 7.5. Over-relaxation is commonly used to speed up the convergence of the Kaczmarz algorithm.

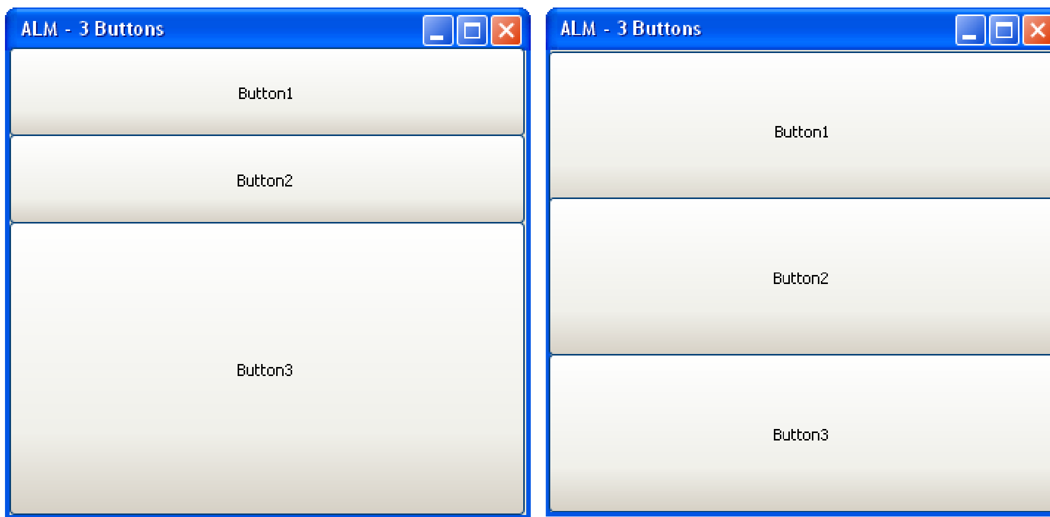
A warm start strategy is applicable to the Kaczmarz algorithm and helps to speed up its convergence. This is described in detail in Chapter 8.

7.5 Least Squares Kaczmarz for Constraints Solving for Attractive GUIs

A common problem in a GUI is that in general a layout cannot allocate its preferred size. In a constraint-based layout a soft constraint can be used to specify the preferred size of a widget. There are various strategies for solving soft constraints. If soft constraints are solved using a linear programming approach then the widget sizes are in general underspecified, and it is undetermined how space is distributed to the widgets. If soft constraints are solved using a least squares approach then available space is distributed in a well-determined way [127].

There has been recent research by Zeidler et al. [127] that shows that a least squares approach yields good aesthetic results for small as well as for large layout sizes. They compared different solving strategies with respect to aesthetics. Their proposed solving strategies were implemented and evaluated using the Auckland Layout Model (ALM), which is the constraint

based layout model described in Chapter 1 of this thesis. They performed a user evaluation where they compared different solving strategies. In this evaluation participants were asked to judge the layouts by their visual appearance, they prefer GUI layouts in which space is distributed according to the preferred size of a layout. They implemented the active set method using a quadratic objective function to get a least squares solution. In this method they try to minimize the sum of deviations from a desired target value. Furthermore, they described that linear and least squares approaches lead to different behaviors when distributing the deviation to the layout items. They analyzed aesthetic aspects for different constraint solving strategies but did not consider the performance of such strategies.



(a) Three buttons using linear programming approach (b) Three buttons using least squares approach

Figure 7.3: Two different solving strategies for a simple three-button layout

This research motivated us to develop such a solving strategy that distributes the available space according to the preferred size of the layout in a GUI, and compare the performance of different constraint solving strategies. If we use our proposed linear constraint solving algorithms for resizing a window in a GUI then a space is not distributed according to the preferred size of the layout. To distribute the space according to the preferred size of the layout, we propose a least squares approach. Figure 7.3 shows the resulting layout solved by using linear programming (linear objective function) and least squares (quadratic objective function) approaches. Figure 7.3 b) shows how a quadratic objective function minimizes the deviation to the preferred item size for each item, not just the sum of deviations of all items.

As a specification in UI layout contains conflicting constraints (the preferred size constraints), the Kaczmarz method will simply diverge. That is why we introduce a least squares approach for solving conflicting constraints in GUIs. For that we first introduce a cooling function in the Kaczmarz algorithm, second we propose an approach for distinguishing hard and soft constraints, and finally we describe solution technique for solving these constraints. The

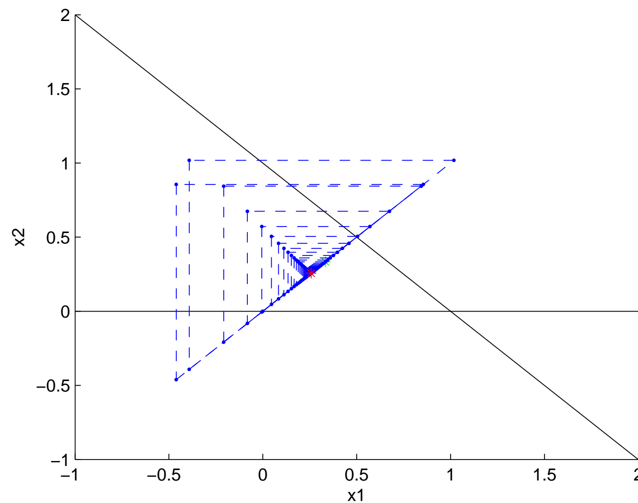


Figure 7.4: Example: Least Squares Kaczmarz method using a cooling function

Least Squares Kaczmarz method using a cooling function is described in the following.

7.5.1 Least Squares Kaczmarz Method using a Cooling Function

The Least Squares Kaczmarz method using a cooling function converges towards a weighted least squares solution for inconsistent systems as the relaxation parameter goes to zero [25]. Given a system of equations $Ax = b$, Least Squares Kaczmarz using a cooling function C^l can be defined as follows:

$$x_{k+1} = x_k + \omega * p * C^l * \frac{(b_i - a_i \cdot x_k)a_i}{\|a_i\|^2}, l = \lfloor k/m + 1 \rfloor \tag{7.7}$$

where x_k is the k-th iterate, $i = (k \mod m)+1$ (for deterministic Kaczmarz), p is the penalty (a weight) assigned to each constraint, a_i is the i-th row of the matrix A, b_i is the i-th component of the right-hand side vector, and $\|a\|$ denotes the Euclidean norm of the vector a. The relaxation parameter for the Kaczmarz algorithm lies inside the interval $0 < \omega < 2$ [95]. In this algorithm, the cooling function is decreased by a factor of C for every m iterations. The value for the cooling function lies inside the interval $0 < C < 1$. The following example illustrates the idea of Least Squares Kaczmarz using a cooling function.

Example 3.
$x_1 = 0$
$x_2 = 0$
$x_1 + x_2 = 1$

Example 3 is an over-determined inconsistent linear system. If we apply Least Squares Kaczmarz using a cooling function, then we eventually get the least squares solution as shown

in Figure 7.4. When solving this example, we set the value for the cooling function to 0.9 as it helps the relaxation parameter to cool down to zero slowly as illustrated in the figure. The relaxation parameter ω is set to 1.9 (over-relaxation) as this helps in speeding up convergence. We also used these parameter values in our experiments for solving randomly generated GUI layout specifications of various sizes, as reported below, and found them to work well.

7.5.2 Distinguishing Hard and Soft Constraints

The minimum constraints are usually hard constraints (see Chapter 4) in a UI layout specification: we give them a very high priority because widgets cannot render if they do not have enough space. The overall size of a layout is fixed and hence should be a hard constraint as well. But the preferred width and height constraints are soft constraints (see Chapter 4 for an explanation of soft constraints), which need to be only satisfied if possible.

Figure 7.3 a) shows a layout solved by a linear programming (linear objective function) approach. All hard constraints are satisfied as expected. The soft constraints for the first two buttons are satisfied exactly, meaning their heights are equal to their preferred heights. The only violated constraint is the preferred height of the third button. As we can see, if we use a linear programming approach then soft constraints are not violated in a uniform way. If we solve these constraints by using the least squares (quadratic objective function) approach then the resulting layout has the desired uniform appearance as shown in Figure 7.3 b). Soft constraints (the preferred size constraints) together with a least squares approach ensure that all widgets have a well-defined size and that there is a unique solution, resulting in an equal distribution of errors among the conflicting constraints.

In ALM (see Chapter 1) all constraints are specified as soft constraints and we assign higher and lower priorities to these constraints. If we use the linear programming approach and there are conflicts between higher and lower priority constraints, then constraints with higher priorities will always be satisfied and constraints with lower priorities will be violated. When we use a least squares approach and there are conflicting constraints then all affected soft constraints are violated to some degree (high-priority constraints less than low-priority constraints, if a weighted least-squares approach is used). To control the violation of soft constraints, we need a conflict resolution strategy where higher priority constraints win over lower priority constraints. The idea is to identify conflicts beforehand, and turn the conflicting constraints with higher priorities into hard constraints, while ensuring the conflicting constraints with lower priorities remain soft constraints. In any case, any non-conflicting constraint can become a hard constraint. This is similar to finding the maximum feasible subset for these constraints.

We propose an approach for distinguishing hard and soft constraints according to priorities that is similar to Algorithm 4 described in Chapter 4. Our approach consists of the following steps that are shown in Algorithm 8. This algorithm starts with an empty set E of enabled

Input: Constraints (**C**)
Output: Hard ("enabled") and soft ("disabled") constraints
1: DISABLE(**C**)
2: SORT(**C**) (by priority)
3: **for** each constraint c in order of priority, descending **do**
4: ENABLE c
5: Solve all enabled constraints (E) using Kaczmarz
6: **if** solution not optimal **then**
7: DISABLE c
8: DISABLE all constraints in C with the same penalty p as c
9: **end if**
10: **end for**

Algorithm 8: Distinguishing hard and soft constraints

constraints (line 1). It then adds constraints incrementally in order of descending priority and solves all enabled constraints using Kaczmarz. Other iterative algorithms could also be used to solve all enabled constraints, but we used the Kaczmarz algorithm because it is a good choice for solving GUI layout problems efficiently, as discussed in Section 7.1. If there is a conflict between higher and lower priority constraints then constraints with lower priority becomes disabled. Iterating through the constraints, it ensures that if a constraint with a certain penalty p was disabled, then all other constraints with the same penalty p are also marked as disabled (line 8). In this way we distribute the error among groups of similar widgets. All disabled constraints become soft constraints and all enabled constraints become hard constraints. In the following section we describe how we solve these constraints.

7.5.3 Solution Technique for Hard and Soft Constraints

After distinguishing hard and soft constraints we apply a hybrid of the original Kaczmarz and Least Squares Kaczmarz for solving these constraints. We treat hard and soft constraints differently. We use the following steps that are shown in Algorithm 9. Iterating through the constraints (line 1), if a constraint is hard (enabled), then it is solved using the original Kaczmarz (without a cooling function). If a constraint is soft (disabled), then it is solved using Least Squares Kaczmarz (with a cooling function), using its penalty p as a weight. The algorithm terminates if the difference between previous and current solution (error e^i) is less than the prespecified tolerance (line 11). The algorithm makes sure that problematic constraints are always solved with least squares, so they will compromise if necessary. If soft constraints of the same penalty p are affected by a conflict, they will share the error equally.

Hard and soft constraints are solved simultaneously because they influence each other. If we solve the hard constraints separately then there are usually infinitely many solutions and we cannot choose the solution that is best for the soft constraints. When solving these constraints together we can choose a feasible solution for the hard constraints while minimizing the error

for the soft constraints.

Input: Constraints (**C**)
Output: Exact Solution for Hard Constraints, Weight Least Squares Solution for Soft Constraints

1: **for** each iteration i **do**
2:     **for** each constraint c **do**
3:         **if** c is hard (enabled) **then**
4:             Apply projection on c using equation 7.2
            //c is soft (disabled)
5:         **else**
6:             Apply projection with a cooling function on c using equation 7.7
7:         **end if**
8:     **end for**
9:     Calculate the error $e^i = x^{previous} - x^{current}$
10:     **if** maximum of the error $e^i \leq$ tolerance **then**
11:         Terminate the algorithm
12:     **end if**
13: **end for**

Algorithm 9: Solution technique for hard and soft constraints

7.6 Experimental Evaluation

In this section we present an experimental evaluation of the proposed algorithms. We conduct three different experiments to evaluate (i) their convergence behaviour, and (ii) their performance in terms of computation time(iii) their ability to detect and resolve conflicts (the quality of a solution).

7.6.1 Methodology

The methodology is the same as in previous Chapters 4 and 6. For all experiments we used the same hardware and test data generator, but instrumentalized the algorithms differently. We used the following setup: a desktop computer with Intel i5 3.3GHz processor and 64-bit Windows 7, running an Oracle Java virtual machine. Layout specifications were randomly generated using the test data generator described in Chapter 3. For each experiment the same set of test data was used. The specification size was varied from 4 to 2402 constraints, in increments of 4 constraints (2 new constraints for positioning and 2 new constraint for the preferred size of a new widget). For each size 10 different layouts were generated, resulting in a total of 6000 different layout specifications. A tolerance of 0.01 was used for solving. For the Kaczmarz method a relaxation parameter of 1.5 was used; for SOR a slightly smaller relaxation parameter of 0.7 had to be used to avoid problems of divergence. For the Least Squares Kaczmarz method with C, a relaxation parameter of 1.9 was used and cooling factor was set to 0.9.

In the first experiment we investigated the convergence behaviour of the algorithms. We measured for each algorithm the number of sub-optimal solutions. A solution is sub-optimal if the error of a constraint (the difference between the right-hand and left-hand side) is not smaller than the given tolerance.

In the second experiment we measured the performance in terms of computation time (T) in milliseconds (ms), depending on the problem size measured in number of constraints (c). Each of the proposed algorithms was used to solve each of the problems of the test data set and the time was measured. As a reference, all the generated specifications were also solved with Matlab's LINPROG solver [115] and LP-Solve [17]. We selected these solvers as LINPROG is widely known for its speed, and LP-Solve has been previously used to solve constraint-based UI layout problems [86]. Additionally, we wanted to test our algorithms against a direct method, so we also included the implementation of QR-decomposition in the Apache Commons Mathematics Library [8], which is a freely available open-source library.

In experiment 3, we evaluated the quality of the solutions, which is given by the integer ι. As explained in Chapter 4, the algorithms should find $\max(\iota)$, so a solution with a larger ι has a better quality. We consider $\bar{\iota}$, the bit-wise negation of ι, as this allows us to differentiate between solutions of different quality more easily. The ι values of solutions for a problem differ usually only in the less significant bits (the most important constraints are usually enabled), whereas $\bar{\iota}$

reflects noteworthy quality differences in the more significant bits.

To calculate $\bar{\iota}$, we set a bit in a bit array of the length of the list of constraints if a constraint is disabled. So in the worst case, $\bar{\iota}$ can become as large as 2^{2402}. Since such numbers are hard to evaluate and interpret, we simplify them by only expressing an ordinal relationship between the solutions of all 5 algorithms for one problem. This is done with integer values $rank$, expressing the ranking of a solution: one is the rank of the solution with the lowest $\bar{\iota}$, and 4 is the rank of the solution with the highest $\bar{\iota}$ for a given problem. Thus, the higher the rank the worse the solution of a solver. In the case of ties, i.e. if two or more algorithms produce the same $\bar{\iota}$, we use the mean of the involved ranks, as is usually done in such situations to preserve the sum of all ranks.

Finally, we use the rank values to compare all algorithms pairwise. We compare two algorithms x and y by testing the distribution of the differences (d)

$$d = rank_x - rank_y$$

for all problems with a *Wilcoxon signed-rank* test on a significance level of $\alpha = 0.001$. If the test accepts the alternative hypothesis that $d < 0$, we conclude that algorithm x produces better results than algorithm y. These results are aggregated over all comparisons. We do not consider QR decomposition in this experiment because it only finds an unweighted least squares solution, i.e. without considering any priorities.

7.6.2 Results

The analysis of results is the same as in previous Chapters 4 and 6 The first experiment demonstrates the convergence behaviour of our algorithms. We found that all algorithms converge, which is obvious since the algorithms were designed to find a solvable subproblem.

In the second experiment we investigated the performance behaviour of all algorithms. To identify the performance trend of the algorithms over c, we defined some regression models (linear, quadratic, log, cubic). We found that the best-fitting model is the polynomial model

$$T = \beta_0 + \beta_1 c + \beta_2 c^2 + \beta_3 c^3 + \epsilon,$$

which gave us a good fit for the performance data.

Key parameters of the models are depicted in Table 7.2; a graphical representation of the models can be found in Figures 7.5 – 7.6. Table 7.1 explains the symbols used.

Figure 7.5 illustrates the performance of Kaczmarz with prioritized IIS detection, Kaczmarz with prioritized grouping constraints, Kaczmarz with CWU, Least Squares Kaczmarz with cooling function and SOR with prioritized grouping constraints. As the graphs indicate, Kaczmarz with prioritized grouping constraints and Kaczmarz with CWU exhibit better performance than

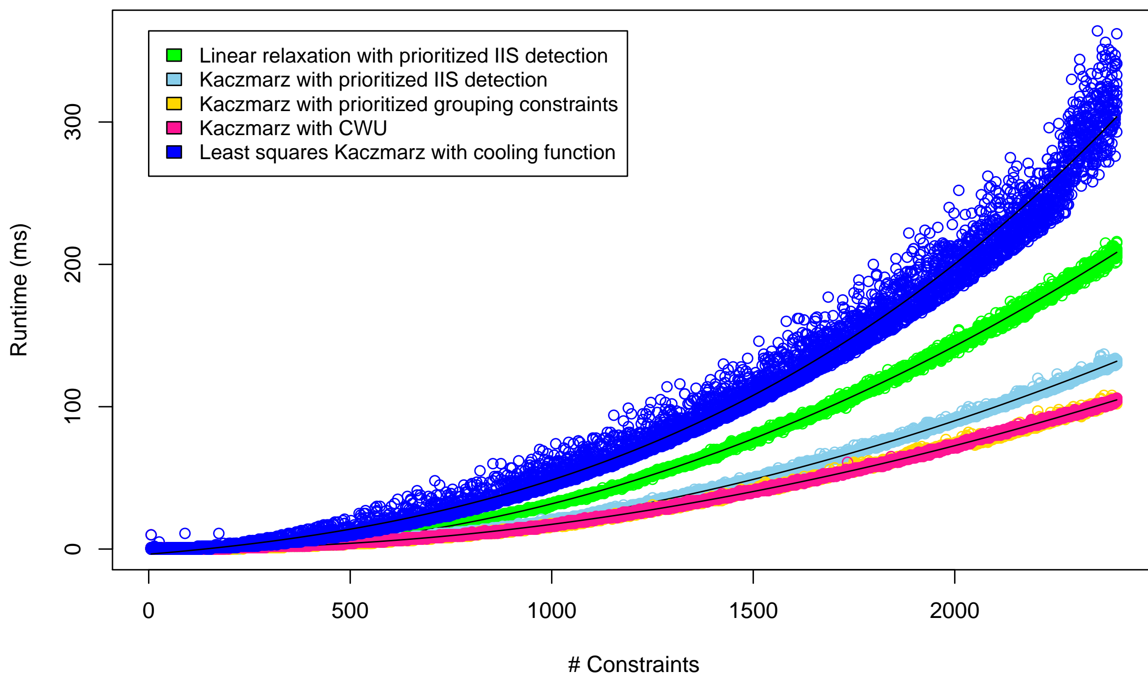


Figure 7.5: Performance comparison of Kaczmarz with prioritized IIS detection, Kaczmarz with prioritized grouping constraints, Kaczmarz with CWU, Least Squares Kaczmarz with cooling function and SOR with random pivot assignment

Kaczmarz with prioritized IIS detection, SOR with prioritized grouping constraints and Least Squares Kaczmarz with cooling function.

Figure 7.6 compares all the aforementioned algorithms to LINPROG, LP-Solve and QR-decomposition. SOR with prioritized grouping constraints is slower than Kaczmarz with prioritized IIS detection. Generally, all our algorithms perform significantly better than LINPROG, LP-Solve and QR-decomposition, especially for bigger problems. Kaczmarz with prioritized grouping constraints exhibits the best runtime behaviour.

SOR with prioritized grouping constraints is slower than Kaczmarz with prioritized IIS detection because the computation of the pivot assignment is more complex and takes longer. This is due to the fact that for SOR with prioritized grouping constraints the pivot assignment only needs to be recomputed for each conflicting constraint. The runtime performance of Kaczmarz with prioritized grouping constraints has the highest volatility. That is due to the fact that the performance of Kaczmarz with prioritized grouping constraints depends on the distribution of conflicting constraints over the list of constraints. If conflicting constraints are close, the algorithm searches only a small fraction of the whole list. If conflicting constraints are almost equally distributed over the list of constraints, the algorithm searches the whole list.

Table 7.3 depicts the results of experiment 3, the comparisons of the algorithms according

Table 7.1: Symbols used for the performance regression model

Symbol	Explanation
β_0	Intercept of the regression model
β_{1-3}	Estimated model parameters
c	Number of constraints
T	Measured time in milliseconds
R^2	Coefficient of determination of the estimated regression models

Table 7.2: Regression models for the different solving strategies

Strategy	β_0	β_1	β_2	β_3	R^2
Pr. grouping constraints / Kacz.	1.143^{***}	$-7.027 \cdot 10^{+03***}$	$2.459 \cdot 10^{-05***}$	$-1.615 \cdot 10^{-09***}$	0.999
CWU / Kacz.	0.9979^{***}	$-6.604 \cdot 10^{-03***}$	$2.172 \cdot 10^{-05***}$	$-7.526 \cdot 10^{-10***}$	0.995
C^l / Least Squares Kacz.	-3.622^{***}	0.02344^{***}	$1.812 \cdot 10^{-05***}$	$1.057 \cdot 10^{-8***}$	0.9367
Pr. IIS detection / rand.	1.035^{***}	$-1.112 \cdot 10^{-02***}$	$4.278 \cdot 10^{-05***}$	$-9.176 \cdot 10^{-10***}$	0.9994
Pr. IIS detection / Kacz.	1.136^{***}	$-7.740 \cdot 10^{+03***}$	$2.723 \cdot 10^{-05***}$	$-5.587 \cdot 10^{-10***}$	0.9994
LINPROG	18.29^{***}	$1.591 \cdot 10^{-04}$	$4.934 \cdot 10^{-05***}$	$1.577 \cdot 10^{-08***}$	0.9367
LP-Solve	-2.491^{***}	$3.924 \cdot 10^{-02***}$	$2.079 \cdot 10^{-04***}$	$1.904 \cdot 10^{-08***}$	0.9900
QR-Decomposition	-37.70^{***}	0.2802^{***}	$-4.009 \cdot 10^{-04***}$	$2.850 \cdot 10^{-07***}$	0.9989

Significance codes: *** $p < 0.001$, Kacz: Kaczmarz

Table 7.3: Comparison of the quality of the solutions produced by the different algorithms (rank 1 = best)

Rank	Strategy
1	Pr. grouping constraints / Kaczmarz, Pr. IIS detection / Kaczmarz
2	Pr. grouping constraints / SOR
3	LINPROG
4	LP-Solve

to the overall solution quality. The pairwise test results showed that algorithms can be ordered totally, i.e. each algorithm is better than all the algorithms ranked below it (i.e. with a bigger rank number). Kaczmarz with prioritized grouping constraints produce the best results. However, Kaczmarz with prioritized IIS detection yields almost as good results as prioritized grouping constraints. The worst solutions are produced by the two simplex solvers, LINPROG and LP-Solve. SOR with prioritized grouping constraints produce mid-quality solutions.

7.6.3 Discussion

The performance results show that the Kaczmarz method with prioritized grouping constraints is the fastest and that QR-decomposition, a direct method, is the slowest for our UI problem.

All proposed algorithms are faster than QR-decomposition, LP-Solve and LINPROG.

The Kaczmarz algorithm with prioritized IIS detection exhibits a better performance than SOR with prioritized IIS detection and Least Squares Kaczmarz with cooling function. A likely factor contributing to this is that for Kaczmarz a slightly larger relaxation parameter can be used than that for SOR . Smaller relaxation parameters may slow down the convergence of iterative methods, potentially requiring more iterations to converge. However, it was not possible to increase the relaxation parameter of SOR as this would cause divergence for some of the problems. Kaczmarz with prioritized grouping constraints requires less computation time than all other strategies, hence it appears to be the most appropriate algorithm.

The runtime of the two linear programming solvers exhibits a much larger variance compared to the purely iterative solvers. One possible reason for this is that for some cases the direct methods used in the linear programming solvers are particularly inefficient, e.g. due to fill-in effects. A smaller variance and hence more predictable runtime is particularly beneficial for the UI layout domain because large changes in runtime can affect the user experience, e.g. when resizing a GUI window interactively.

It is natural that LINPROG and LP-Solve are outperformed by the newly introduced algorithms in terms of solution quality, since they resolve conflicts differently. If a high-priority constraint needs to be violated, these approaches will not simply "abandon" the constraint, as our algorithms would do. Instead, they still try to minimize this violation, even if this comes at the cost of violating many lower-priority constraints. This means a worse $\bar{\iota}$ value is generated.

7.7 Summary

This chapter proposed new algorithms for using the Kaczmarz method for solving constraint-based UI layout problems. We have presented the following contributions:

- We developed algorithms to resolve conflicts in over-determined specifications by using soft constraints.
- We extended Kaczmarz for solving linear inequality and soft constraints for GUI.
- We extended Kaczmarz for solving constraints for GUIs by generating balanced layouts: it can help in making GUIs attractive.
- We presented an experimental evaluation that shows that the proposed algorithms find feasible subproblems and outperform modern linear programming solvers.

The work presented in this chapter lays a foundation for the application of iterative methods for solvers of constraint-based UIs.

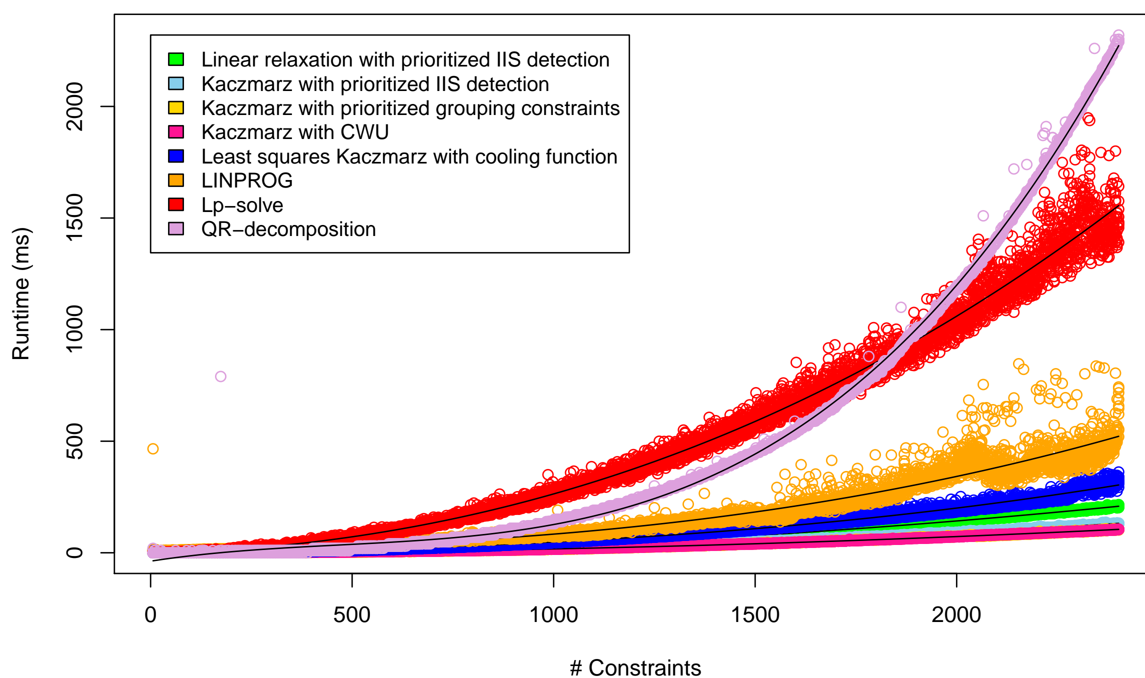


Figure 7.6: Performance comparison of the best solving strategies with LINPROG, LP-Solve, QR-decomposition and prioritized IIS detection using random pivot assignment

8

Speeding Up Kaczmarz and Successive Over-RelaxatioS with a Warm-Start Strategy

8.1 Introduction

Many computer programs have graphical user interfaces (GUIs), which need good layout to make efficient use of the available screen real estate. Most GUIs do not have a fixed layout, but are resizable and able to adapt themselves. Constraints are a powerful tool for specifying adaptable GUI layouts: they are used to specify a layout in a general form, and a constraint solver is used to find a satisfying concrete layout, e.g. for a specific GUI size. The constraint solver has to calculate a new layout every time a GUI is resized or changed, so it needs to be efficient to ensure a good user experience. One common approach for constraint solvers is based on the Gauss-Seidel algorithm and Successive Over-Relaxation (SOR) and the previous chapter described another frequently used approach: the Kaczmarz algorithm.

Our observation is that a solution after resizing or changing is similar in structure to the previous solution. Thus, we hypothesized that we could increase the computational performance of the Kaczmarz and an SOR-based constraint solvers if we reuse the solution of a previous layout to warm-start the solving of a new layout. In this chapter we report on experiments to test this hypothesis experimentally for three common use cases: big-step resizing, small-step

resizing and constraint change. In our experiments, we measured the solving time for randomly generated GUI layout specifications of various sizes. For all three cases we found that the performance is improved if an existing solution is used as a starting solution for a new layout.

The remainder of the chapter is organized as follows. We disscuss related work in Section 8.2 and then explain the warm start strategy in Section 8.3. We test our hypothesis experimentally with randomly generated GUIs in Section 8.4. To give the context for our experiments we give some required background on constraint-based GUIs and iterative solvers in Chapter 2, and draw our conclusions in Section 8.5.

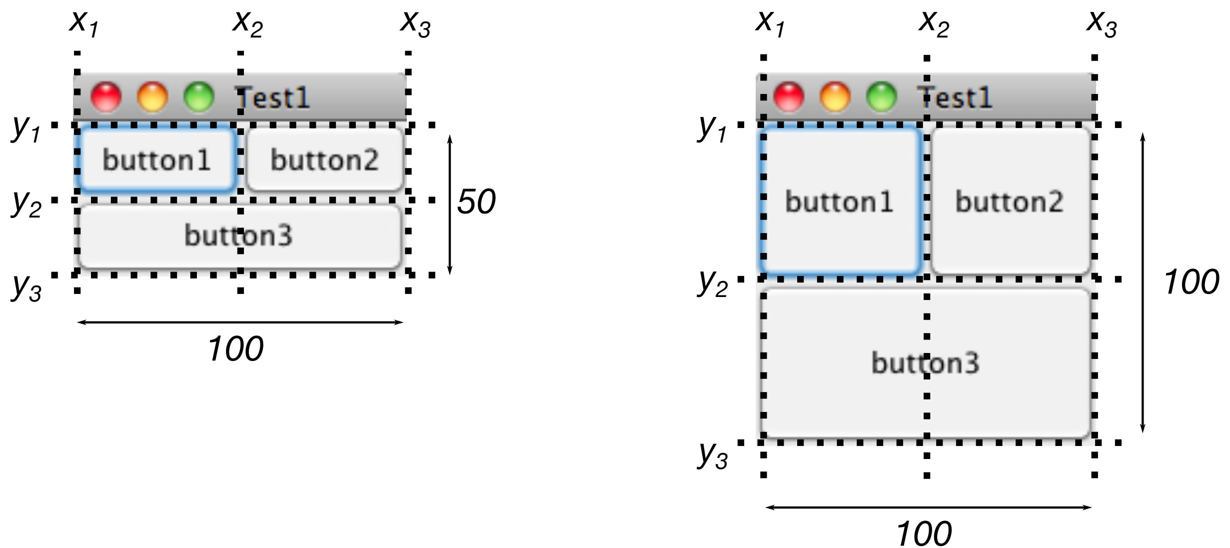
8.2 Related Work

Approaches related to warm start strategies have been proposed in numerous previous works [48, 54,93]. Warm-start strategies with interior-point methods have been applied by authors [76,124] for linear programming problems. They [41, 120] proposed a warm-start strategy based on an active set method for solving quadratic programming problems. A warm-start strategy based on an interior point method for solving quadratic programs in predictive control is proposed by [111].

Lessard et al. analyzed computational speed and the use of warm starting (where they used the most recent estimate for initializing the iterative scheme) of different iterative methods for large systems [81]. These methods include multigrid, preconditioned conjugate-gradient, and several new variants of these methods. They identified the best algorithm with the warm-start strategy in terms of computational time. They [122] used warm starting approaches to speed up gradient projection for sparse reconstruction (GPSR) and iterative shrinkage/thresholding (IST) algorithms, in these algorithms the warm-started approach is used to reduce the computational cost from an advanced starting point for the sparse reconstruction by separable approximation (SpaRSA) algorithmic framework for solving large-scale optimization problems. Other methods of accelerating convergence by using warm start techniques in iterative solution procedures are [42,44,104]. However, none of the above discussed algorithms applies Kaczmarz and SOR-based methods for constraint-based UI layout and therefore none explicitly exploit the sparsity property of constraint-based UI problems.

8.3 Warm-Start Strategy

A GUI layout specification has to be solved whenever the conditions under which a GUI is displayed or the GUI itself change. Most GUIs can be resized, e.g. to adapt to different screen sizes or to let the user choose an appropriate size dynamically. Sometimes GUIs need to be changed dynamically to adjust to content of different sizes. Each time a GUI is resized or changed, the existing GUI layout specification is changed and a new specification is created. However, the



	$x_3 = 100$, $y_3 = 50$		$x_3 = 100$, $y_3 = 100$
Button 1	Hard: $x_2 - x_1 \geq 40$, $y_2 - y_1 \geq 20$	Button 1	Hard: $x_2 - x_1 \geq 40$, $y_2 - y_1 \geq 20$
	Soft: $x_2 - x_1 = x_3/2$, $y_2 - y_1 = y_3/2$		Soft: $x_2 - x_1 = x_3/2$, $y_2 - y_1 = y_3/2$
Button 2	Hard: $x_3 - x_2 \geq 40$, $y_2 - y_1 \geq 20$	Button 2	Hard: $x_3 - x_2 \geq 40$, $y_2 - y_1 \geq 20$
	Soft: $x_3 - x_2 = x_3/2$, $y_2 - y_1 = y_3/2$		Soft: $x_3 - x_2 = x_3/2$, $y_2 - y_1 = y_3/2$
Button 3	Hard: $x_3 - x_1 \geq 40$, $y_3 - y_2 \geq 20$	Button 3	Hard: $x_3 - x_1 \geq 40$, $y_3 - y_2 \geq 20$
	Soft: $x_3 - x_1 = x_3$, $y_3 - y_2 = y_3/2$		Soft: $x_3 - x_1 = x_3$, $y_3 - y_2 = y_3/2$

Figure 8.1: A GUI constraint specification before and after resizing

new specification is similar to the previous one because the widgets and their relations typically stay the same. Usually, only some size parameters change. For example, Figure 8.1 shows a GUI before and after resizing with the corresponding constraint specifications. Only the height constraint at the beginning of the specification is changed.

As a consequence, constraint solvers for GUI layout usually have to solve specifications that are similar to the specifications that have been solved previously. For that reason, it seems plausible that the previous solution is a good initial value for the iterative solving process – something that is known as a warm-start strategy. This leds us to the following hypothesis: *Using a previous solution of a GUI constraint specification to warm-start the Kaczmarz and SOR solvers reduces the solving time.*

We tested this hypothesis by considering three common use cases where GUIs are changed during runtime. The first is small-step resizing, where the GUI size is changed by a small amount, e.g. when it is resized by a user dynamically. The second is big-step resizing, where the GUI size is changed by a larger amount, e.g. when the GUI size is maximized. And third, changes of several constraints, e.g. when the sizes of labels are adjusted for a different language. The solving times when using Kaczmarz and SOR with and without a warm-start strategy was compared for the three use cases, using randomly generated layout specifications of different

sizes.

During application runtime, GUIs need to be adapted to changing conditions such as the available GUI size. This is done by changing some of the constraints, typically a small number. For example, the overall size of the GUI is typically specified by constraints: one for the width and one for the height. When the GUI size changes, only these two constraints need to be adjusted, as shown in Figure 8.1. Another typical situation where constraints need to be changed is when preferred sizes change. For example, if the language settings are changed in an application, the preferred sizes of textual labels have to adjust to the new language.

8.4 Experiment

In this section, we evaluate the use of warm-starting for GUI layout problems using the Kaczmarz algorithm and SOR. We tested specifically the effect of warm-starting a constraint solver on the performance in terms of computation time.

8.4.1 Methodology

We conducted the experiments with our implementation of the Kaczmarz and SOR methods for GUI layout, which uses random pivot assignment and Prioritized IIS Detection as a conflict resolution strategy. We used two versions of that solver: the first version started every solver run with an initial solution $x = (0, \ldots, 0)$, the second version started every solver run with the optimal solution from the previous run $x = x^*_{\text{prev}}$.

We evaluated the following three use cases:

1. *Small-step resizing*: The width and height of the window was randomly changed by a value in between 0 and 3 pixels.

2. *Big-step resizing*: The width and height of the GUI window was randomly changed by a value between 4 and 3000 pixels.

3. *Constraint change*: 10 per cent of all constraints of a GUI were randomly changed.

Small-step resizing occurs in practice when a window is continuously resized by dragging a window border. Big-step resizing occurs when a GUI is initially loaded on different screens, when a GUI is switched to or from full-screen mode, or when the orientation of a screen is changed. Constraint changes as in use case 3 occur, for example, when several preferred sizes change as a result of changing the language of an application.

Layout specifications were randomly generated using the parameterized algorithm described in Chapter 3. The problem size was varied from 0 to 201 areas. For each area 4 constraints are added, which specify the position of the area in the layout. Additionally, a specification

needs 4 constraints to define the size of the window. We therefore started with a problem of 4 constraints and ended with a problem of 808 constraints. For each size, 10 random layouts were evaluated. For each of the three use cases, each of these random layouts was changed 20 times, and the solving time was measured. A linear relaxation parameter of 0.7 and a tolerance of 0.01 were used for solving. The measurements were performed on a desktop computer with Intel i5 3.3GHz processor and 64-bit Windows 7 running an Oracle Java virtual machine.

Table 8.1: Symbols used for the performance regression model

Symbol	Explanation
β_0	Intercept of the regression model
β_{1-3}	Estimated model parameters
c	Number of constraints
T	Measured time in milliseconds
R^2	Coefficient of determination

To compare the performance of both versions of the solver we used a regression model

$$T = f(c) + \epsilon$$

and examined the estimated model visually and numerically. See Table 8.1 for an explanation of the symbols used.

8.4.2 Results

To identify the performance trend of the solvers, we tried different regression functions f (linear, quadratic, log, cubic). We found that the best fitting model is the polynomial model

$$T = \beta_0 + \beta_1 c + \beta_2 c^2 + \beta_3 c^3 + \epsilon.$$

Key parameters of the regression models are depicted in Table 8.2. A graphical representation of the measurements and the models can be found in Figures 8.2, 8.3 and 8.4. The results suggest a better performance of the solver with the warm-start strategy for all three use cases.

The variance of the measured runtime differs noticeably for both approaches. It is smaller for the rather small changes in small-step resizing, and bigger for the big-step resizing and constraint change use cases. Especially for constraint change, this indicates that some problems with a lot of conflicts were generated, which require more iterations and hence a longer runtime. The variance of the measured runtime differs for both warm start and cold start approaches. It looks as if the former is smaller, which is another advantage of the warm-start approach. It is somewhat astonishing that the experiments showed the warm start strategy had only a relatively

Table 8.2: Regression models for solvers with and without warm-start strategy

Strategy	β_0	β_1	β_2	β_3	R^2
Ssr with warm start / SOR	$6.508 \cdot 10^{4***}$	$-8.940 \cdot 10^{2***}$	23.55^{***}	$7.576 \cdot 10^{-4***}$	0.999
Ssr w/o warm start / SOR	$4.104 \cdot 10^{4***}$	68.00^{***}	26.05^{***}	$5.971 \cdot 10^{-3***}$	0.999
Ssr with warm start / Kacz.	$1.864 \cdot 10^{-2***}$	$-2.435 \cdot 10^{-4***}$	$7.181 \cdot 10^{-6***}$	$2.076 \cdot 10^{-4***}$	0.9992
Ssr w/o warm start / Kacz.	$1.173 \cdot 10^{-2***}$	$2.870 \cdot 10^{-5***}$	$8.129 \cdot 10^{-6***}$	$4.991 \cdot 10^{-9***}$	0.9937
Bsr with warm start / SOR	$3.186 \cdot 10^{4***}$	-89.32^{***}	13.42^{***}	$3.202 \cdot 10^{-3***}$	0.996
Bsr w/o warm start / SOR	$1.208 \cdot 10^{4***}$	$3.729 \cdot 10^{2***}$	18.24^{***}	$4.921 \cdot 10^{-3***}$	0.999
Bsr with warm start / Kacz.	$2.407 \cdot 10^{-2***}$	$-2.781 \cdot 10^{-4***}$	$6.981 \cdot 10^{-6***}$	$1.811 \cdot 10^{-9***}$	0.9957
Bsr w/o warm start / Kacz.	$3.570 \cdot 10^{-3***}$	$2.796 \cdot 10^{-4***}$	$8.231 \cdot 10^{-6***}$	$6.731 \cdot 10^{-9***}$	0.9829
Cc with warm start / SOR	$1.074 \cdot 10^{5***}$	$-1.289 \cdot 10^{3***}$	21.88^{***}	$-2.721 \cdot 10^{-4***}$	0.976
Cc w/o warm start / SOR	$1.231 \cdot 10^{5***}$	$-1.648 \cdot 10^{3***}$	30.17^{***}	$-2.837 \cdot 10^{-4***}$	0.962
Cc with warm start / Kacz.	$1.864 \cdot 10^{-2***}$	$-2.435 \cdot 10^{-4***}$	$7.181 \cdot 10^{-6***}$	$2.076 \cdot 10^{-9***}$	0.9992
Cc w/o warm start / Kacz.	$1.173 \cdot 10^{-2***}$	$2.870 \cdot 10^{-5***}$	$8.129 \cdot 10^{-6***}$	$4.991 \cdot 10^{-9***}$	0.9937

Significance codes: *** $p < 0.001$, Ssr: Small-step resizing, Bsr: Big-step resizing, Cc: Constraint changes

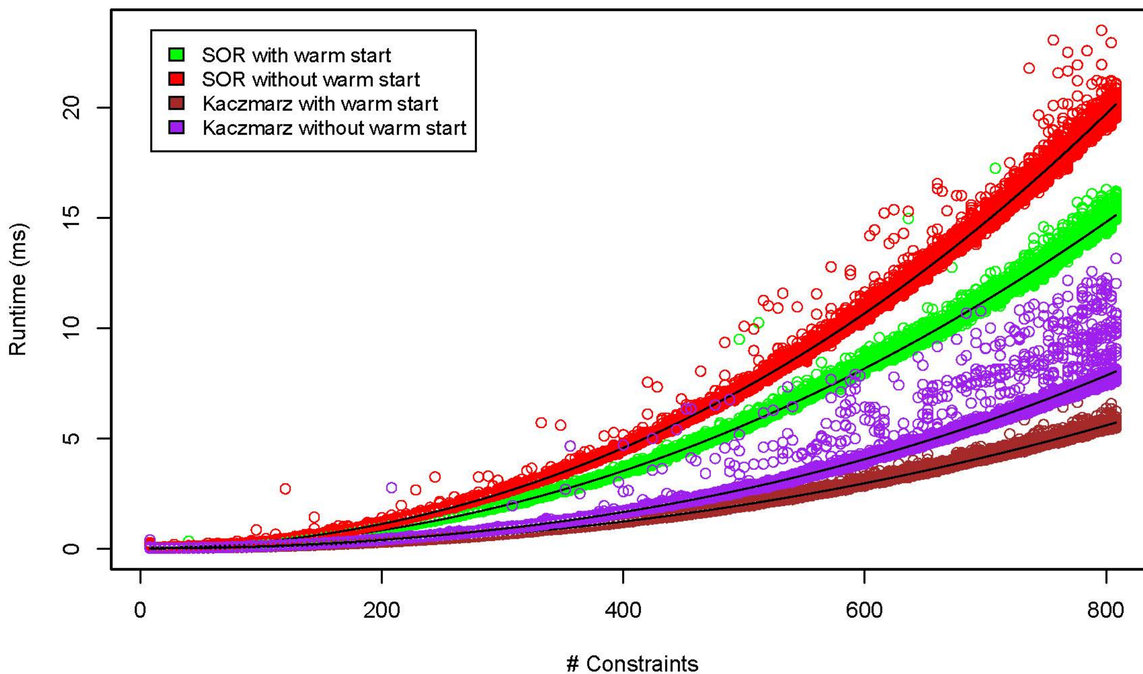


Figure 8.2: Small-step resizing performance results

small effect. One reason could be the use of the random pivot selector. Since it selects pivot elements randomly, it can select pivot elements which let the solution deviate strongly from the initial solution before it actually converges towards the new solution. Another reason could be that the changes in the specification – even though they are fairly small – drastically change the solution in some cases. This can be, for example, due to conflict resolution. Some constraints

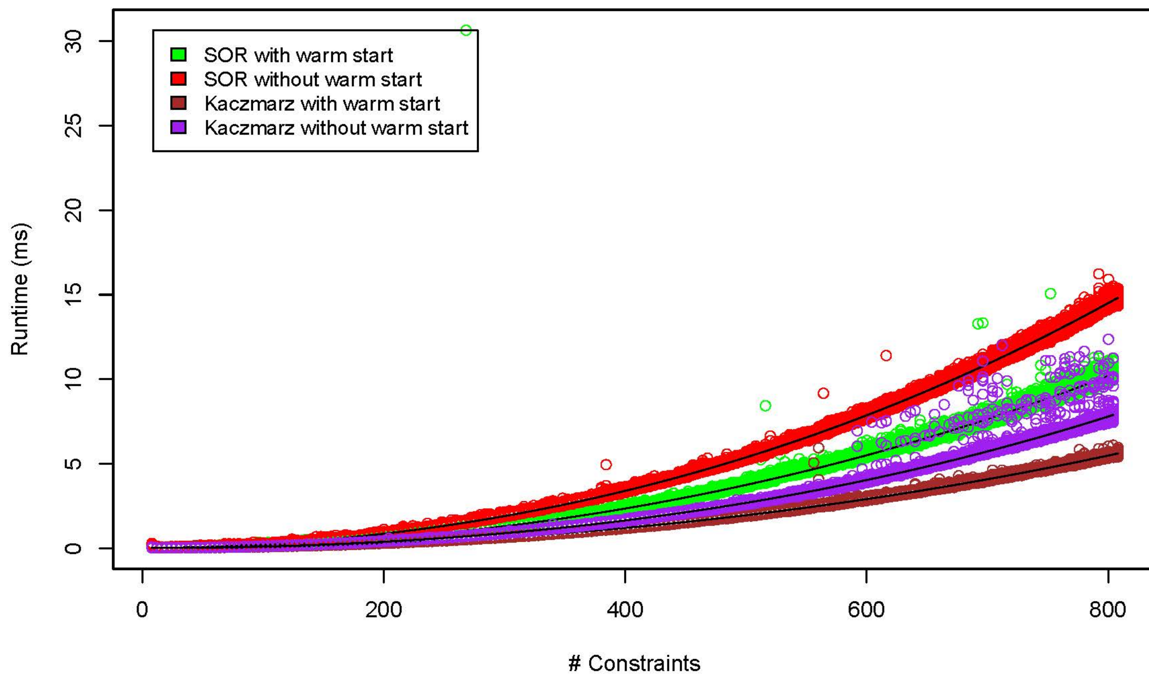


Figure 8.3: Big-step resizing performance results

that were not satisfied with the old specification, can become satisfiable and suddenly have an effect on the solution after the specification is changed. Similarly, small changes in the specification can lead to new conflicts and hence disabling of constraints.

The effects of the warm-start strategy are comparable for all three use cases, but are the strongest for small-step resizing. This is convenient, as speed is of particular importance for the small-step resizing use case. Small-step resizing is typically done interactively by the user, and for a good user experience the GUI should react to such resizing in real-time.

8.5 Summary

In constraint-based GUIs with dynamic behaviour, the specification that represents the layout of the GUI is often changed, e.g. when a window is resized. These changes are usually small, resulting in specifications that are very similar. Since the specifications are similar, one can expect also the results to be similar. Therefore, we evaluated the use of a warm-start strategy to improve the efficiency of Kaczmarz and SOR-based constraint solvers for GUIs. Three common use cases were evaluated with randomly generated GUI layouts: small-step resizing, big-step resizing, and random changes of several constraints.

We found that the Kaczmarz and SOR-based methods with a warm-start strategy indeed

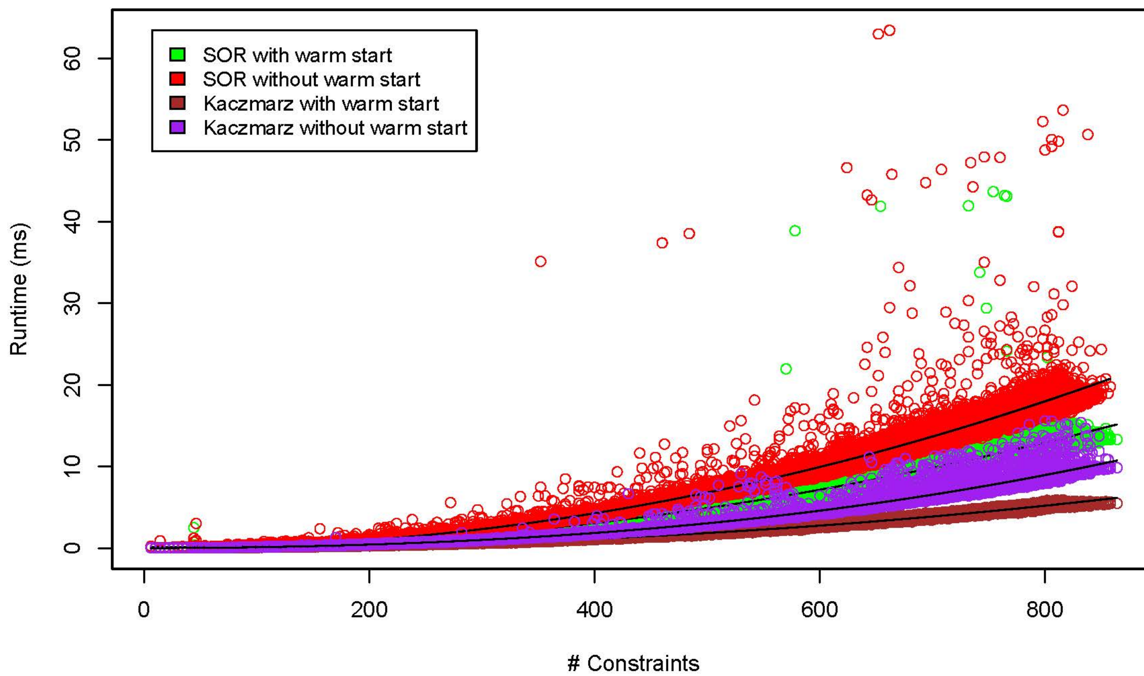


Figure 8.4: Constraint change performance results

exhibit a better runtime behaviour than a solver without a warm-start strategy. Implementing a warm-start strategy in such solvers does not introduce additional computational effort, as existing values are simply reused. It is therefore advisable to equip Kaczmarz and SOR-based GUI layout solvers with a warm-start strategy.

However, we also found that the effect of a warm-start strategy is weaker than we expected. Possible reasons for this are the random pivot assignment and the Prioritized IIS Detection conflict resolution strategy used in the experiment.

9
Framework Design

The concepts, algorithms and techniques presented in the previous chapters are implemented in several research prototypes, which are based on the Auckland Layout Model (ALM) [86]. ALM is a constraint-based layout manager with ports to several programming languages and environments. Besides others, a Java-based implementation is available that uses the popular Simplex-based solver Lp-Solve [17] to solve GUI layout problems, which is available online [1] and is used in several other research projects. More details are given in Section 9.1.

The algorithms implemented in this thesis can be considered as extensions to ALM, which aim at replacing the Simplex method with indirect methods. Their implementations are contained in the package *linsolve* and described in Section 9.2. An overall summary of different components involved in ALM and *linsolve* is provided here as a class diagram in Figure 9.1.

9.1 Auckland Layout Model (ALM)

There are various implementations of constraint-based layout managers available of whom the most prominent is the Cocoa API of Apple's Mac OS X[2]. Beside this there exists some academic implementations. One of them is the Auckland Layout Model (ALM) [86] which is freely available and already the basis of several research projects [128]. ALM allows to describe GUIs in an algebraic way which makes it possible to create layouts in a more abstract manner. A more

[1] `http://lpsolve.sourceforge.net`

[2] Cocoa Auto Layout Guide, 2012 `http://developer.apple.com`

detailed description of ALM is given in Chapter 1. Below is a description of all important ALM classes and interfaces as illustrated in Figure 9.1.

9.1.1 ALMLayout

`ALMLayout` is a general class for creating layouts on the screen. It implements a layout manager and can be used by Java applications which need to use ALM as a layout manager. The `LayoutManager` class is a Java interface for GUI layout, it defines the interface for classes that know how to lay out widgets. In addition to standard methods for a layout manager, `ALMLayout` provides a set of methods to enable users to benefit from ALM specific features. For example, the $addConstraint()$ method allows users to add a customized set of constraints to the layout. At the end, when a user completes the design of the layout, everything is translated to a linear system and stored in a `LayoutSpec` object. This linear system is solved by linear solvers which will be described in Section 9.2.

9.1.2 Variable, XTab and YTab

The `XTab` class represents a vertical grid line and the `YTab` class represents a horizontal grid line. They extend the `Variable` class. They are used by the methods of `ALMLayout`.

9.1.3 Row and Column

The `Row` class represents a row defined by two y-tabs and the `Column` class represents a column defined by two x-tabs.

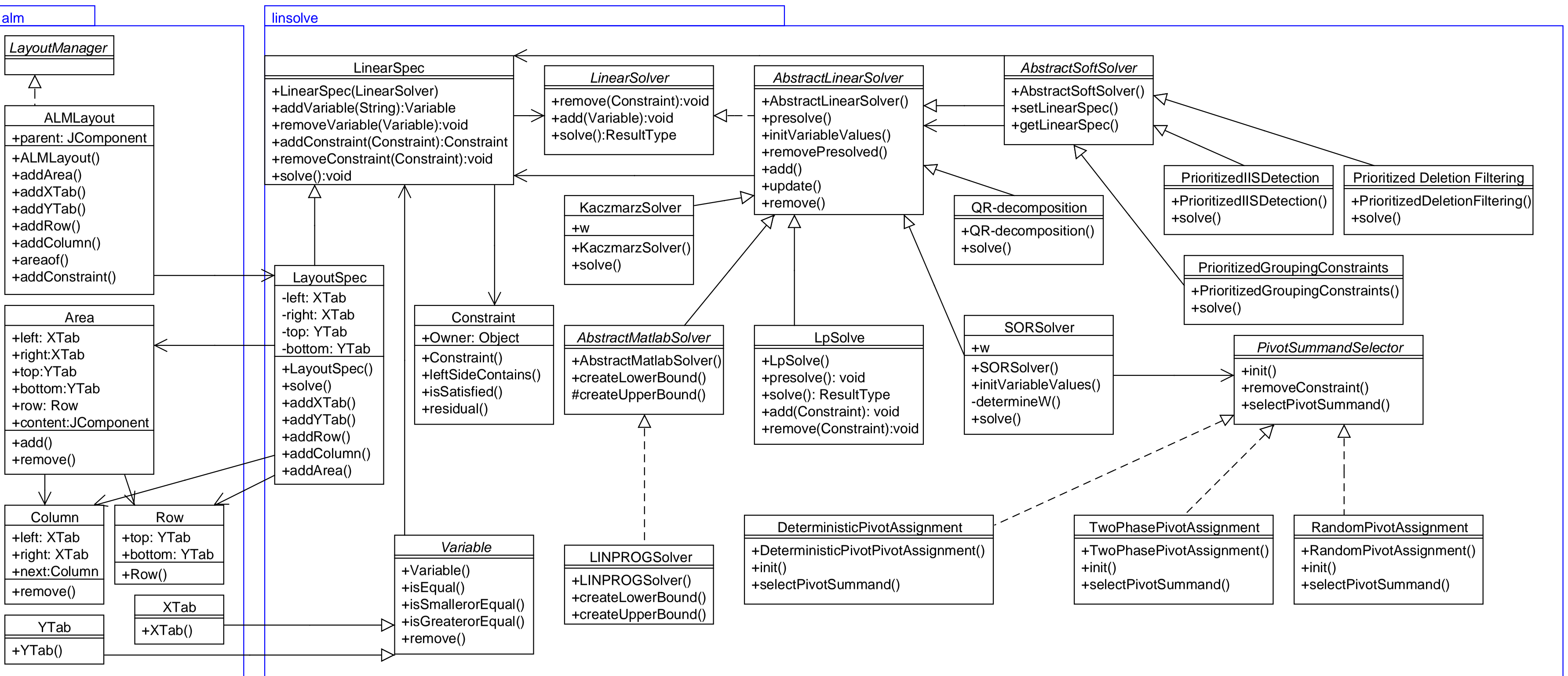


Figure 9.1: Class Diagram of different components involved in alm and linsolve.

9.2 linsolve

There are different solvers implemented in the linsolve package for solving linear systems of constraints created by ALM. Their classes are illustrated in Figure 9.1.

9.2.1 LinearSpec

`LinearSpec` is a class for accumulating constraints. Constraints are generated by different classes, e.g. `Area` and `Constraint`. Solvers have access to the `LinearSpec` class. It contains the linear system which should be solved by a solver when its $solve()$ method is called.

9.2.2 LayoutSpec

The `LayoutSpec` class contains information about layout components. Users create areas using method $addArea()$ in this class. Each layout has an associated `LinearSpec` which contains the linear system that describes the layout.

9.2.3 Constraint

The `Constraint` class represents a linear constraint. The constraint can be in form of a equality or an inequality. A penalty can be associated to the constraint which specifies its priority. Higher priority constraints have less chance to get violated.

9.2.4 LinearSolver

The `LinearSolver` is an interface for all solvers. The $add(Variable)$ method adds a variable in `LinearSolver` and $remove(Constraint)$ method tells a solver to remove a constraint from this solver which is important because some solvers manage their own representation of constraints. The $solve()$ method tries to solve the linear problem and return the result of the solving attempt.

9.2.5 AbstractLinearSolver

The `AbstractLinearSolver` is an abstract implementation of a linear solver. All solvers extend this abstract class. The $initVariableValues()$ method initialise all variables with 0 if they do not contain a previous value.

If a cached simplified version of the problem exists, it is used instead of `LinearSpec`. The $removePresolved()$ method is used to remove a cached presolved model, if existent.This is automatically done each time after the model has been changed to avoid an old cached presolved

model getting out of sync. The $add()$ method adds a constraint to this solver. Some solvers keep their own set of constraints, $remove()$ method removes a constraint from this solver.

The $update()$ method tells a solver to update a constraint in this solver because some solvers manage their own representation of constraints.

The `KaczmarzSolver`, `LpSolve`, `SORSolver`, `QR-decomposition` and `LINPROG` are implementations of solvers used in Chapter 4 and Chapter 7.

9.2.6 AbstractSoftSolver

The `AbstractSoftSolver` is an abstract class for conflict resolution strategies such as `PrioritizedIISDetection`, `PrioritizedGroupingConstraints` and `PrioritizedDeletionFiltering`(see Chapter 4 for details). The actual solver is injected via the provided constructor. The class gets a `LinearSpec` and makes sure that the constraints in the specification are treated as soft constraints.

9.2.7 SOR and PivotSummandSelector

`SORSolver` is a concrete implementation of `AbstractSoftSolver` and contains the implementation of a SOR solver (see Chapter 2 for details). SOR and its foundation the Gauss-Seidel Algorithm are not capable of solving non-square systems. To nevertheless apply SOR to such systems a pivot assignment has to be done.

`PivotSummandSelector` is used for a pivot assignment which is an interface of `RandomPivotAssignment`, `DeterministicPivotAssignment` and `TwoPhasePivotAssignment`. They are explained in detail in Chapter 3.

9.2.8 AbstractMatlabSolver

The `AbstractMatlabSolver` is an abstract class for `LINPROGSolver`, which uses Matlab code to solve constraint-based GUIs. `LINPROG` is an implementation only available in Matlab. Calls to `LINPROG` have first to be translated to proper Matlab calls. This is done with `AbstractMatlabSolver`. To connect to Matlab it uses the `matlabcontrol` library. It basically requires Matlab to be installed together with Matlab's Optimization toolbox. The main contribution of MatlabSolver is the generation of Matlab-commands, reading and writing back the calculated results. The intended use of the class is to compare the runtime of other solvers with the runtime of Matlab solvers.

10
Conclusion and Future Work

This chapter concludes the thesis and points out future directions. Section 10.1 lists the major achievements of this thesis and Section 10.2 discusses possible directions for future work.

10.1 Achievements

The primary focus of this research was to design efficient algorithms for solving constraint-based GUI layout specifications. We highlight the major achievements of our work below.

- **Extending SOR for Non-Square Matrices.** SOR is one of the most commonly used iterative methods for solving linear constraint problems. Non-square matrices occur in GUI layout specifications, this is a problem for the common SOR method, which assumes that the problem matrix is square and has a non-zero diagonal. The standard SOR algorithms choose the pivot variable on the diagonal of the coefficient matrix. In the case of square matrices with non-zero diagonals, this is an easy way to ensure that every constraint has a pivot variable, and that every variable is chosen once so that its value can be approximated. The common SOR algorithms usually assume that a pivot assignment has been performed and that the chosen pivot elements are placed on the diagonal of the problem matrix. In square matrices, this can always be achieved by simple matrix transformations.

  In this first contribution, we extended the SOR method to solve non-square matrices. We proposed three pivot assignment algorithms for solving non-square matrices. In the first

algorithm, pivot elements were selected pseudo-randomly. In the second algorithm, pivot elements were selected deterministically by optimizing certain selection criteria. In the third algorithm, the best constraint for a variable was selected and the best variable for a constraint was selected. The problem of pivot assignment in the case of non-square matrices and the two algorithms were explained in detail in Chapter 3. These results have been published at the ICTAI'12 conference [71].

- **Extending SOR for Soft Constraints.** Linear problems including UI layout may contain conflicting constraints. SOR will simply not converge if a specification contains conflicting constraints. To deal with conflicting constraints, we introduced *soft constraints*. In contrast to the usual *hard* constraints, which cannot be violated, soft constraints may be violated if no other solution can be found. Using only soft constraints has the advantage that a problem is always solvable, which cannot be guaranteed if hard constraints are used. In the second contribution, we proposed three conflict resolution algorithms for solving systems of prioritized linear constraints with the SOR method. In the first algorithm, non-conflicting constraints are successively added in descending order of priority. The second algorithm starts with all constraints and successively removes conflicting constraints in ascending order of priority. The third algorithm is a mixture of both and adapts the binary search algorithm to the problem of searching the best conflict-free subproblem. These algorithms were explained in detail in Chapter 4. Some of these results have been published at the ICTAI'12 conference [71].

- **An Optimization of SOR.** If SOR is applied to linear systems, the ordering in which constraints are solved affects the convergence behaviour of the algorithm. A bad ordering can slow down the convergence of an algorithm whereas a good ordering can speed it up. To overcome this problem, we proposed an algorithm to reorder the sequence of constraints specification in an optimal way. Our contribution consists of, first, a metric to measure the optimality of a constraint sequence and, second, a Simulated Annealing based algorithm, which optimizes the order of constraints that was explained in detail in Chapter 5. These results have been accepted at the IJCSM'14 journal [73].

- **A New Optimization Technique for SOR.** One of the disadvantages of iterative methods is that convergence is not guaranteed for general matrices. We showed empirically that if we choose Constraint-Wise Under-relaxation (CWU), then SOR convergence is improved for GUI layout problems.

- **Extending Kaczmarz with Soft Constraints.** Computing the pivot assignment for solving non-square matrices can make Successive Over-Relaxation (SOR) slow. To overcome this, we considered the Kaczmarz method in Chapter 7, which does not need any pivot assignment for solving non-square matrices. These results have been published at IC-

TAI'13 conference [74]. Moreover, the least squares Kaczmarz method is introduced for solving attractive GUIs.

- **Speeding Up Kaczmarz and SOR with a Warm-Start Strategy.** One approach to layout is to use constraints to describe the GUI and let a constraint solver find a satisfying layout. One type of constraint solver is based on the Gauss-Seidel algorithm and Successive Over-Relaxation (SOR) and another type of solver is Kaczmarz. The most current GUIs are not of fixed size but resizable. A constraint solver has to find a new layout every time a GUI is resized, which can be time consuming. Our observation was that a solution after resizing was similar in structure to a solution before resizing. We hypothesized that we could increase the computational performance of Kaczmarz and SOR-based constraint solvers if we reused the solution of a previous layout to warm start the solving of a new layout. In Chapter 8, we explained a warm start strategy to speed up Kaczmarz and SOR for GUI layouts. The results of this research have been published at the ICDIM'13 conference [72]. Together the contributions demonstrate that iterative methods can efficiently be used for solving constraint-based GUIs. With the algorithms presented in this thesis, it is now possible to bring the benefits of solving sparse matrices efficiently with iterative methods to the domain of UI layout.

10.2 Future Directions

This section describes possible future directions in which this research could be extended. Time constraints prevented us from exploring these areas ourselves.

- There is room for improvement in the deterministic pivot assignment algorithm. The results of the experiment indicate that an optimal pivot assignment can have a significant effect on the speed of convergence. Currently, deterministic pivot assignment only takes the influence of coefficients of constraints into account. The inferior performance of this assignment compared to a purely random one indicates that there are other factors that have an effect on convergence. One such factor is the order of the constraints during solving.

- Some applications in constraint-based UIs could benefit from the formulation of non-linear constraints. Integrating the solving of non-linear constraints into the framework of the Gauss-Seidel method would extend the application domain of our algorithms. Some non-linear constraints can be solved with Gauss-Seidel and SOR.

- Other iterative algorithms using an optimal relaxation parameter for user interface layouts and general linear problems could also be evaluated.

- The warm starts strategy can also be used to a wider class of problems. Besides the reuse of existing values, there are also other types of pre processing techniques for iterative solvers available, which could improve the convergence behaviour of SOR solvers for constraint-based GUIs. Examples include coefficient matrix reordering or the application of pre conditioners. Their applicability and effect on the efficiency of SOR solvers for constraint-based GUIs should also be tested.

  Other future directions include testing all our proposed algorithms on general linear problems and presenting comparisons between various iterative algorithms.

10.3 Applications of Iterative Methods

This section surveys real world applications of our extended iterative methods, one could use these methods outside of User Interface domain.

10.3.1 Saddle Point Problems

Saddle point problems are encountered in a wide variety of fields such as computer science and engineering. A number of applications can be found in the real world where saddle point problems naturally arise, for example computational fluid dynamics [51,103], image reconstruction [57], constrained optimization [121] and economics [9]. Saddle point problems naturally arise when a certain quantity has to be minimized subject to certain constraints. The solution of saddle point problems is a prominent area in the numerical research field due to its specific character and its appearance in many engineering fields such as fluid and solid mechanics [23].

Large and sparse problems usually occur in saddle point problems. There are various methods that have been developed for solving these problems recently. While, saddle point systems have widely different structural and sparsity properties and come in different sizes. One reason why Iterative methods are of interest for solving these problems is that they are usually large and sparse.

10.3.2 Computerized Tomography

Computerized Tomography (CT) was introduced in the 1970s for the purpose of diagnostic radiology, but since then various applications of it have been developed and have become popular(e.g. electro-magnetic geotomography). Computed tomography has played an important role in medicine and industrial applications [61,75]. As a result of this, continuous efforts are made in order to improve image reconstruction algorithms. CT is concerned with reconstruction of a function from its line or plane integrals. From a mathematical point of view, such problems

are known as inverse problems. Computerized Tomography gives rise to large and sparse linear systems of equations.

These linear systems of equations which occurring in computer tomography are both under-determined and over-determined. Various methods have been developed for solving these problems, the most well known method is image reconstruction from projections. Stationary iterative methods can be applied for solving the problems that appear in Computerized Tomography(CT), because these methods are more efficient for solving large and sparse problems compared to direct methods.

10.4 Reflections

In this section I provide some personal opinions and thoughts related to my PhD experience. In the end I truly enjoyed the process of writing a thesis, though it was difficult sometimes to motivate myself if the work was very challenging. For example when doing performance testing of my proposed algorithms for large problems, nonobvious memory issues on the testbed created problems, and it took long time to figure out this issue.

The motivation behind the project was in the beginning quite challenging. I found it more interesting when, by working with example problems, the motivation became more clear to me. There is also further work opportunities on the theory part of some of the proposed algorithms. Some extensions e.g. of the strategies to deal separately with soft constraints are yet to be explored, but could not be included due to a lack of time.

Looking back, I can now say that that this research project allowed me to gain a deep understanding of iterative algorithms and application of these algorithms on constraint-based GUI layout. I hope that my proposed algorithms for solving constraints for GUI layout can serve as a platform for future research.

Bibliography

[1] Pratik Agarwal and Edwin Olson. Variable reordering strategies for slam. *In Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems(IROS)*, 2012.

[2] Shmuel Agmon. The relaxation method for linear inequalities. *Canadian Journal of Mathematics*, pages 382–392, 1954.

[3] E. Amaldi. From finding maximum feasible subsystems of linear systems to feed-forward neural network design. *PhD thesis no. 1282, Dèpartement de Mathè-matiques, cole Polytechnique Fèdèrale de Lausanne, Switzerland*, 1994.

[4] E. Amaldi, M. Bruglieri, and G. Casale. A two-phase relaxation-based heuristic for the maximum feasible subsystem problem. *Computers and Operations Research*, pages 1465–1482, 2008.

[5] H. M. Anita. *Numerical Methods for Scientist and Engineers*. Birkhauser, 2002.

[6] Carlos Ansǿtegui, María Luisa Bonet, and Jordi Levy. Solving (weighted) partial maxsat through satisfiability testing. *In Procedings of the 20th International Conference on Theory and Applications of Satisfiability Testing*, 5584:427–440, 2009.

[7] Carlos Ansǿtegui, Maria Luisa Bonet, and Jordi Levy. A new algorithm for weighted partial maxsat. *In Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence*, 2010.

[8] Apache Software Foundation. Commons Math: The Apache Commons mathematics library, 2012.

[9] K. J. Arrow and R. M Solow. *Gradient methods for constrained maxima with weakened assumptions*. Stanford University Press, Stanford, CA, 1958.

[10] Owe Axelsson. *Iterative Solution Methods*. Cambridge Uni. Press, 1996.

[11] G. J. Badros, A. Borning, and P. J Stuckey. The cassowary linear arithmetic constraint solving algorithm. *In ACM Transactions on Computer-Human Interaction*, 8(4):267–306, 2001.

[12] R.R. Bakker, F. Dikker, F. Tempelman, and P.M. Wogmim. Diagnosing and solving over-determined constraint satisfaction problems. *International Joint Conference on Artificial Intelligence*, pages 276–281, 1993.

[13] Mokhtar S. Bazaraa, John J. Jarvis, and Hanif D. Sherali. *Linear Programming and Network Flows*. Wiley, 4th edition, 2009.

[14] M. Benzi, D.B. Szyld, and A. Van Duin. Ordering for incomplete factorization pre-conditioning of nonsymmetric problems. *SIAM Journal on Scientific Computing*, pages 1652–1670, 1999.

[15] M. Benzi and M. Tuma. Orderings for factorized sparse approximate inverse precondi-tioners. *SIAM Journal on Scientific Computing*, pages 1851–1868, 2000.

[16] Michele. Benzi. Preconditioning techniques for large linear systems: A survey. *Journal of Computational Physics*, 182:418–477, 2002.

[17] Michel Berkelaar, Jeroen Dirks, Kjell Eikland, Peter Notebaert, and Juergen Ebert. Lp-solve: A (mixed integer) linear programming problem solver, 2012.

[18] D. L Berre. Sat4jmaxsat, a satisfability library for java: `http://sat4j.org/ (2008).`

[19] E. Bodewig. *Matrix Calculus*. Interscience, New York, Amsterdam, 2nd edition, 1959.

[20] Alan Borning, Richard J. Anderson, and Bjørn N. Freeman-Benson. Indigo: A local propagation algorithm for inequality constraints. *ACM Symposium on User Interface Software and Technology*, pages 129–136, 1996.

[21] Alan Borning, Bjorn Freeman-Benson, and Molly Wilson. Constraint hierarchies. *Lisp and Symbolic Computation*, 5(3):223–270, 1992.

[22] Alan Borning, Kim Marriott, Peter Stuckey, and Yi Xiao. Solving linear arithmetic con-straints for user interface applications. In *Proceedings of the Annual ACM Symposium on User Interface Software and Technology*, pages 87–96, 1997.

[23] F. Brezzi and M. Fortin. Mixed and hybrid finite element methods. *Springer Series in Computational Mathematics, Springer, New York*, 15, 1991.

[24] Richard L. Burden and J. Douglas Faires. *Numerical Analysis*. Bob Pirtle, 2005.

[25] Y. Censor, P. P. B. Eggermont, and D. Gordon. Strong underrelaxation in kaczmarz method for inconsistent systems. *Numerische Mathematik*, pages 83–92, 1983.

[26] Yair Censor. Row-action methods for huge and sparse systems and their applications. *SIAM review*, 23(4):444–466, 1981.

[27] L. Cesari. Sulla risoluzione dei sistemi di equazioni lineari per approssimazioni successive. *Atti Accad. Nazionale Lincei R. Classe Sci. Fis. Mat. Na*, 25(422), 1937.

[28] Steven C. Chapra and Raymond Canale. *Numerical Methods for Engineers*. McGraw-Hill, Inc., New York, USA, 5th edition, 2006.

[29] John W. Chinneck. Fast heuristics for the maximum feasible subsystem problem. *Informs Journal of Computation*, pages 210–223, 2001.

[30] John W. Chinneck and E. Dravnieks. Locating minimal infeasible constraint sets in linear programs. *ORSA Journal on Computing*, pages 157–168, 1991.

[31] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 24th National Conference*, pages 157–172, 1969.

[32] J. Dancis. The optimal w is not best for the sor iteration method. *Linear Algebra and Its Applications*, pages 819–845, 1991.

[33] George B. Dantzig. *Linear Programming and Extensions*. Princeton Uni. Press, 11th edition, 1998.

[34] Biswa Nath Datta. *Numerical Linear Algebra And Applications*. Cole, 1995.

[35] Feng Ding and Tongwen Chen. Iterative least-squares solutions of coupled sylvester matrix equations. *Systems and Control Letters*, 54:95–107, 2005.

[36] I. S. Duff and J. Koster. On algorithms for permuting large entries to the diagonal of a sparse matrix. *Technical Report, Rutherford Appleton Laboratory, Oxon, UK and ENSEEIHT-IRIT, Toulouse, France*, 1999.

[37] L.C. Dutto. The effect of ordering on preconditioned gmres algorithm. *International Journal for Numerical Methods in Engineering*, pages 457–497, 1993.

[38] P. P. B. Eggermont, G. T. Herman, and A. Lent. Iterative algorithms for large partitioned linear systems, with applications to image reconstruction. *Linear Algebra and Its Applications*, pages 37–67, 1981.

[39] M. Eiermann and R.S. Varga. Is the optimal w best for the sor iteration method. *Linear Algebra and its Applications*, 182:257–277, 1993.

[40] Tommy Elfving. Block-iterative methods for consistent and inconsistent linear equations. *Numerical Mathematics*, 35(1):1–12, 1980.

[41] H.G. Ferreau, H.J.and Bock and M. Diehl. An online active set strategy to overcome the limitations of explicit mpc. *International Journal of robust and Nonlinear Control*, pages 816–830, 2008.

[42] R. Fletcher. *Practical Methods of Optimization*. Wiley-Interscience, 1987.

[43] E. Florez, M.D. Garcia, L. Gonzalez, and G. Montero. The effect of orderings on sparse approximate inverse preconditioners for non-symmetric problems. *Journal of Advances in Engineering Software*, pages 611–619, 2002.

[44] Anders Forsgren. On warm starts for interior methods. In *System Modelling and Optimization*, pages 51–66, 2005.

[45] L. V. Foster. Modifications of the normal equations method that are numerically stable. *In Numerical Linear Algebra, Digital Signal Processing and Parallel Algorithms*, pages 501–512, 1991.

[46] Freeman-Benson, John Maloney, and Alan Borning. An incremental constraint solver. *Communications of the ACM*, 33(1):54–63, 1990.

[47] Eugene C. Freuder. Partial constraint satisfaction. *In International Joint Conference on Artificial Intelligence*, pages 278–283, 1989.

[48] R. M. Freund. A potential function reduction algorithm for solving a linear program directly from an infeasible warm start. *Mathematical Programming*, pages 441–466, 1987.

[49] Aurél Galántai. *Projectors and Projection Methods*, volume 6. Springer, 2003.

[50] Kamila Ghidetti, Lucia Catabriga, Maria Claudia Boeres, and Maria Cristina Rangel. A study of the influence of sparse matrices reordering algorithms on krylov-type preconditioned iterative methods. *Mecánica Computacional*, pages 2323–2343, 2010.

[51] R. Glowinski. *Numerical Methods for Nonlinear Variational Problems*. Springer Series in Computational Physics, New York, 1984.

[52] J. L. Goffin. The relaxation method for solving systems of linear inequalities. *Mathematics of Operations Research*, 5(3):388–414, 1980.

[53] G.H. Golub and C.F. Van Loan. *Matrix Computations*. Johns Hopkins Uniersity Press, 1996.

[54] J. Gondzio and J.-P. Vial. Warm start and epsilon-subgradients in the cutting plane scheme for block-angular linear programs. *Computational Optimization and Applications*, pages 17–36, 1999.

[55] R. Gordon, R. Bender, and G. T. Herman. Algebraic reconstruction techniques (ART) for three-dimensional electron microscopy and X-ray photography. *Journal of Theoretical Biology*, 29:471–481, 1970.

[56] O. Guieu and J. W. Chinneck. Analyzing infeasible mixed-integer and integer linear programs. *INFORMS Journal on Computing*, pages 63–77, 1999.

[57] E.L. Hall. *Computer Image Processing and Recognition*. Academic Press, New York, 1979.

[58] Martin Hanke and Wilhelm Niethammer. On the acceleration of kaczmarz method for inconsistent linear systems. *Linear Algebra and Its Applications*, 130:83–98, 1990.

[59] Michael T. Heath. *Scientific Computing, An Introductory Survey*. McGraw-Hill, 1997.

[60] Federico Heras, Javier Larrosa, and Albert Oliveras. Minimaxsat: a new weighted max-sat solver. *International Conference on Theory and Applications of Satisfiability Testing*, pages 41–55, 2007.

[61] G.T. Herman. Image reconstruction from projections: the fundamentals of computed tomography. *Journal of Applied Mathematics and Mechanics*, 9(3):446–448, 1982.

[62] Magnus R. Hestenes and Eduard Stiefel. Methods of Conjugate Gradients for Solving Linear Systems. *Journal of Research of the National Bureau of Standards*, 49:409–436, 1952.

[63] H. Hosobe and S. Matsuoka. A foundation of solution methods for constraint hierarchies. *Constraints*, 8(1):41–59, 2003.

[64] H. Hosobe, S. Matsuoka, and A. Yonezawa. Generalized local propagation: A framework for solving constraint hierarchies. *In Proceedings of the Second International Conference on Principles and Practice of Constraint Programming*, 1996.

[65] Hiroshi Hosobe. A scalable linear constraint solver for user interface construction. In *Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming*, pages 218–232, 2000.

[66] Hiroshi Hosobe. A simplex-based scalable linear constraint solver for user interface applications. In *Proceedings of International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 793–798, 2011.

[67] Hiroshi Hosobe, Ken Miyashita, Shin Takahashi, Satoshi Matsuoka, and Akinori Yonezawa. Locally simultaneous constraint satisfaction. In *Proceedings of the Second International Workshop on Principles and Practice of Constraint Programming*, pages 51–62, 1994.

[68] A. J. Hughes Hallett and Laura Piscitelli. Simple reordering techniques for expanding the convergence radius of first-order iterative techniques. *Journal of Economic Dynamics and Control*, pages 1319–1333, 1998.

[69] A.J. Hughes Hallett and P.G.Fisher. On economic structures and model solution methods. *Oxford Bulletin of Economics and Statistics 52*, pages 317–330, 1990.

[70] C. G. J. Jacobi. Über ein leichtes Verfahren, die in der Theorie der Säkularstörungen vorkommenden Gleichungen numerisch aufzulösen. *Crelle's Journal*, pages 51–94, 1846.

[71] Noreen Jamil, Johannes Müller, Christof Lutteroth, and Gerald Weber. Extending linear relaxation for user interface layout. *In Proceedings of 24th International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 1–8, 2012.

[72] Noreen Jamil, Johannes Müller, Christof Lutteroth, and Gerald Weber. Speeding up sor solvers for constraint-based guis with a warm-start strategy. *International Conference on digital information management(ICDIM)*, pages 268–273, 2013.

[73] Noreen Jamil, Johannes Müller, Christof Lutteroth, and Gerald Weber. Constraints reordering to speed up successive over-relaxation for constraint-based user interface layouts. *International Journal of Computing Science and Mathematics (IJCSM)*, 2014.

[74] Noreen Jamil, Johannes Müller, Deanna Needell, Christof Lutteroth, and Gerald Weber. Extending kaczmarz algorithm with soft constraints for user interface layout. *In Proceedings of 25th International Conference on Tools with Artificial Intelligence (ICTAI)*, 2013.

[75] L. Jian, C. Litaoa, L.and Penga, S. Gia, and W. Zhifang. Rotating polar coordinate art-applied in industrial image reconstruction. *NDT and Ein*, 40(4):333–336, 2007.

[76] E. John and E.A. Yildirim. Implementation of warm-start strategies in interior-point methods for linear programming in mixed dimension. *Computational Optimization and Applications*, pages 151–183, 2008.

[77] Ulrich Junker. Quickxplain: preferred explanations and relaxations for over-constrained problems. *In Proceedings of the 19th national conference on Artifical intelligence*, pages 167–172, 2004.

[78] S. Kaczmarz. Angenäherte auflösung von systemen linearer gleichungen. *Bulletin International de l'Acadmie Polonaise des Sciences et des Lettres. Classe des Sciences Mathmatiques et Naturelles. Srie A, Sciences Mathmatiques*, pages 355–357, 1937.

[79] W. Kahan. Gauss-seidel methods of solving large systems of linear equations. *Doctoral thesis, University of Toronto*, 1858.

[80] Stefan Kunis and Holger Rauhut. Random sampling of sparse trigonometric polynomials, ii. orthogonal matching pursuit versus basis pursuit. *Journal Foundations of Computational Mathematics*, 8(6):737–763, 2008.

[81] Laurent Lessard, Matthew West, Douglas MacMynowski, and Sanjay Lall. Warm-started wavefront reconstruction for adaptive optics. *Journal of the Optical Society of America*, 25(5):1147–1155, 2008.

[82] D. Leventhal and A. S. Lewis. Randomized methods for linear constraints: Convergence rates and conditioning. *Mathematics of Operations Research*, 35(3):641–654, 2010.

[83] Chu Min Li, Felip Many, Nouredine Ould Mohamedou, and Jordi Planes. Exploiting cycle structures in max-sat. *SAT, Lecture Notes in Computer Science, Springer*, 5584:467–480, 2009.

[84] Han Lin, Kaile Su, and Chu Min Li. Within-problem learning for efficient lower bound computation in max-sat solving. *In International Conference on Theory and Applications of Satisfiability Testing*, pages 351–356, 2008.

[85] Martin Lukasiewycz, Michael Glaß, Felix Reimann, and Jürgen Teich. Opt4J - A Modular framework for meta-heuristic optimization. In *Proceedings of the Genetic and Evolutionary Computing Conference*, Dublin, IL, 2011.

[86] Christof Lutteroth, Robert Strandh, and Gerald Weber. Domain specific high-level constraints for user interface layout. *Constraints*, 13(3), 2008.

[87] Christof Lutteroth and Gerald Weber. Modular specification of gui layout using constraints. *In Proceedings of the 19th Australian Conference on Software Engineering. Washington, DC, USA: IEEE Computer Society*, pages 300–309, 2008.

[88] O. Mangasarian. Misclassification minimization. *Journal of Global Optimization*, pages 309–323, 1994.

[89] Vasco M. Manquinho, Joo P. Marques Silva, and Jordi Planes. Algorithms for weighted boolean optimization. *In Procedings of the 20th International Conference on Theory and Applications of Satisfiability Testing*, pages 495–508, 2009.

[90] M. J. Maron. *Numerical-Analysis*. Collier Macmillan, 1982.

[91] Kim Marriott, Sitt Chen Chok, and Alan Finlay. A tableau based constraint solving toolkit for interactive graphical applications. In *Proceedings of the 4th International Conference on Principles and Practice of Constraint Programming*, pages 340–354, 1998.

[92] Pedro Meseguer, Noureddine Bouhmala, Taoufik Bouzoubaa, Morten Irgens, and Martí Sánchez. Current approaches for solving over-constrained problems. *Constraints*, 8(1):9–39, 2003.

[93] J. E. Mitchell and B. Borchers. Solving real-world linear ordering problems using a primal dual interior point cutting plane method. *Annals of Operations Research*, pages 253–276, 1996.

[94] Th. Motzkin and I. J. Schoenberg. The relaxation method for linear inequalities. *Canadian Journal of Mathematics*, pages 393–404, 1954.

[95] F. Natterer. *The Mathematics of Computerized Tomography*. Wiley, 1986.

[96] D. Needell and R. Ward. Two-subspace projection method for coherent overdetermined linear systems. *Journal of Fourier Analysis and Applications*, 19(2):256–269, 2013.

[97] Deanna Needell. Randomized Kaczmarz solver for noisy linear systems. *BIT*, 50(2):395–403, 2010.

[98] Deanna Needell and Joel A Tropp. Paved with good intentions: Analysis of a randomized block kaczmarz method. *Linear Algebra and Its Applications*, 2013. Forthcoming.

[99] M. Pfetsch. Branch and cut for the maximum feasible subsystem problem. *SIAM Journal on Optimization*, pages 21–38, 2008.

[100] Marc Pirlot. General local search methods. *European Journal of Operational Research*, 92(3):493–511, 1996.

[101] C. Popa. Least squares solution of overdetermined inconsistent linear systems using kaczmarz relaxation. *International Journal of Computer Mathematics*, pages 86–102, 1995.

[102] C. Popa and R. Zdunek. Kaczmarz extended algorithm for tomographic image reconstruction from limited-data. *Mathematics and Computers in Simulation*, 65:579–598, 2004.

[103] Quarteroni and A. Valli. *Numerical Approximation of Partial Differential Equations*, volume 23. Springer Series in Computational Mathematics, 1994.

[104] Xiang Ren and Zhouchen Lin. Linearized alternating direction method with adaptive penalty and warm starts for fast solving transform invariant low-rank textures. *International Journal of Computer Vision*, pages 1–14, 2013.

[105] Daniel Ruiz. A scaling algorithm to equilibrate both rows and columns norms in matrices. *Technical Report, Rutherford Appleton Laboratory, Oxon, UK and ENSEEIHT-IRIT, Toulouse, France*, 2001.

[106] D. B. Russel. On obtaining solutions to the navier-stokes equations with automatic digital computers. *Ministry of Aviation, Aeronautical Research Council, Reports and Memoranda*, 1963.

[107] Y. Saad. Iterative methods for sparse linear systems. *SIAM, Philadelphia, PA*, page 112, 2003.

[108] Youcef Saad and Martin H Schultz. Gmres: a generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, 7(3):856–869, 1986.

[109] A. Bhatti Saeed and A. Bhatti Naeem. *Numerical Analysis*. Shahryar, 2008.

[110] Michael Sannella. Skyblue: a multi-way local propagation constraint solver for user interface construction. In *Proceedings of the 7th annual ACM symposium on User interface software and technology*, pages 137–146, 1994.

[111] A. Shahzad, E. C. Kerrigan, and G. A. Constantinides. A warm-start interior-point method for predictive control. *In UKACC International Conference on CONTROL*, pages 949–954, 2010.

[112] R.V. Southwell. *Relaxation Methods in Theoretical Physics*. Clarendon Press, Oxford, 1846.

[113] J. C. Strikwerda. Iterative methods for the numerical solution of second order elliptic equations with large first order terms. *SIAM Journal on Scientific and Statistical Computing*, 1:119–130, 1980.

[114] Thomas Strohmer and Roman Vershynin. A randomized Kaczmarz algorithm with exponential convergence. *Journal of Fourier Analysis and Applications*, 15(2):262–278, 2009.

[115] Jeff Stuart. Linprog:http://www.mathworks.com/matlabcentral/fileexchange/97-linprog/.

[116] I. Sutherland. Sketchpad: a man-machine graphical communication system. *In Proceedings of IFIP Spring Joint Computer Conference*, 1963.

[117] Hamdy A. Taha. *Operations Research: An Introduction*. Mcmillan Publishing, 1992.

[118] M Tamiz, S J Mardle, and D F Jones. Resolving inconsistency in infeasible linear programmes. *Technical Report, School of Mathematical Studies, University of Portsmouth, U.K*, 1995.

[119] M Tamiz, S J Mardle, and D F Jones. Detecting iis in infeasible linear programmes using techniques from goal programming. *Computers and Operations Research*, pages 113–119, 1996.

[120] Y. Wang and S. Boyd. Fast model predictive control using online optimization. *IEEE Transactions on Control Systems Technology*, pages 267–278, 2010.

[121] S.J. Wright. *Primal Dual Interior Point Methods*. Siam, Philadelphia, PA, 1997.

[122] S.J. Wright, R.D. Nowak, and M. A T Figueiredo. Sparse reconstruction by separable approximation. *IEEE Transactions on Signal Processing*, 57(7):2479–2493, 2009.

[123] Shiming Yang and Matthias K. Gobbert. The optimal relaxation parameter for the sor method applied to the poisson equation in any space dimensions. *Applied Mathematics Letters*, pages 325 – 331, 2009.

[124] E.A. Yildirim and S.J. Wright. Warm-start strategies in interior-point methods for linear programming. *SIAM Journal on Optimization*, pages 782–810, 2002.

[125] Yasuhiro Yoshioka, Hiroshi Masuda, and Yoshiyuki Furukawa. A constrained least squares approach to interactive mesh deformation. In *Proceedings of the IEEE International Conference on Shape Modeling and Applications*, pages 23–33, 2006.

[126] David M. Young, Jr. *Iterative Solution of Large Linear Systems*. Academic Press, 1971.

[127] Clemens Zeidler, Christof Lutteroth, and Gerald Weber. Constraint solving for beautiful user interfaces: How solving strategies support layout aesthetics. *In Proceedings of CHINZ*, pages 23–32, 2012.

[128] Clemens Zeidler, Christof Lutteroth, and Gerald Weber. An evaluation of stacking and tiling features within the traditional desktop metaphor. In *Human-Computer Interaction INTERACT*, pages 702–719. 2013.

[129] A. Zouzias and N. M. Freris. Randomized extended Kaczmarz for solving least-squares. *SIAM Journal on Matrix Analysis and Applications*, May 2012. Forthcoming.