2012 IEEE 24th International Conference on Tools with Artificial Intelligence

Extending Linear Relaxation for User Interface Layout

Noreen Jamil, Johannes Müller, Christof Lutteroth, Gerald Weber

Department of Computer Science
University of Auckland
Private Bag 92019, Auckland, New Zealand
{njam031, jmue933}@aucklanduni.ac.nz, {lutteroth, gerald}@cs.auckland.ac.nz

***Abstract*—Linear relaxation is a common method for solving linear problems as they occur in science and engineering. In contrast to direct methods such as Gauss-elimination or QR-factorization, linear relaxation is inherently efficient for problems with sparse matrices as they are often encountered, for instance, in the application domain of constraint-based UI layout. However, the linear relaxation method 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 the UI layout problem.**

In this paper we extend linear relaxation to non-square matrices and soft constraints, and identify pivot assignment as the major issue to overcome in this process. We propose two algorithms for pivot assignment: random pivot assignment, and a more complex deterministic pivot assignment algorithm. Compared to the standard pivot assignment, which selects the elements on the diagonal of the problem matrix as pivot elements, these algorithms make the solving process more robust and make it possible to solve non-square matrices. Furthermore, we propose two algorithms for solving specifications containing soft constraints: constraint insertion and constraint removal. With these algorithms, it is possible to prioritize constraints. That is, if there are conflicting constraints in a specification as is commonly the case for UI layout, only the constraints with lower priority are violated to resolve the conflict.

The performance and convergence of the proposed algorithms are evaluated empirically using randomly generated UI layout specifications of various sizes. The results show that our best linear relaxation algorithm performs significantly better than that of LP-Solve, which is a well-known efficient linear programming solver, and QR-decomposition.

***Index Terms*—UI layout, linear relaxation, soft constraints, non-square matrices.**

I. Introduction

Linear problems are encountered in a variety of fields such as engineering, mathematics and computer science. 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 [1].

Many linear problems are sparse, i.e. most linear coefficients in the corresponding matrix are zero so that the number of non-zero coefficients is $O(n)$ with n being the number of variables [2]. 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 [3].

Iterative methods do not spend processing time on coefficients that are zero. Direct methods, in contrast, 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 we therefore may weaken the sparsity property and may have to deal with more coefficients, which makes processing slower. Therefore iterative, indirect methods are often faster than naive direct methods in such cases.

We are concerned with a domain where sparse problems occur frequently, namely user interface (UI) layout. Section II describes the common properties of this domain, and delineates the solving approaches that have been proposed for it. The contributions of this paper are motivated by and were evaluated for the UI layout problem.

One of the most common iterative methods used to solve sparse linear systems is linear relaxation. Starting with an initial guess, it 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. Linear relaxation with its variations and properties is discussed in Section III. Despite its efficiency for sparse systems, linear relaxation is currently not widely used for UI layout, for the reasons explained in the following.

A common property of many linear problems in UI layout is that the matrices corresponding to these linear problems are non-square. For example, when specifying UI layouts with linear constraints, there are generally more constraints than variables. This is possible because of soft constraints and the presence of inequalities. Both reasons are discussed later. Also, constraints in user interfaces are often not in general position because of many coefficients being $1, 0$ or -1. Finally, constraints are often generated and may be generated repeatedly in one specification. Therefore there is a reasonable likelihood that a UI specification contains redundant, non-conflicting constraints.

This leads to the problem of *pivot assignment*: the problem of choosing a pivot variable for each constraint so that a iterative method converges. The standard linear relaxation

1082-3409/12 $26.00 © 2012 IEEE
DOI 10.1109/ICTAI.2012.132

algorithms choose for each constraint the pivot variable on the diagonal of the coefficient matrix. However, in the general case the diagonal approach has several problems:

1) Diagonal elements may be zero, making them infeasible as pivot elements.
2) Diagonal elements may be small compared to other elements in the same row of a matrix, making them a bad choice that may cause the solving process to diverge.
3) If there are more constraints than variables, then not every constraint contains an element on the diagonal of the coefficient matrix.
4) If there are less constraints than variables, then not every variable is chosen at least once. This means that its value cannot be adapted and the method might not find a solution.

As a first contribution, we describe how the linear relaxation method can be extended to deal with these problems. We propose two pivot assignment algorithms that can be used with any problem matrix, regardless of its shape or diagonal elements. A pivot assignment algorithm has to ensure that every constraint has a pivot variable, and that every variable is chosen at some stage. The first algorithm selects pivot elements pseudo-randomly. The second algorithm selects pivot elements deterministically by optimizing certain selection criteria. The overall problem of pivot assignment and the two algorithms are explained in detail in Section IV.

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

To deal with conflicts, *soft constraints* need to 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. Using only soft constraints has the advantage that a problem is always solvable, which cannot be guaranteed if hard constraints are used.

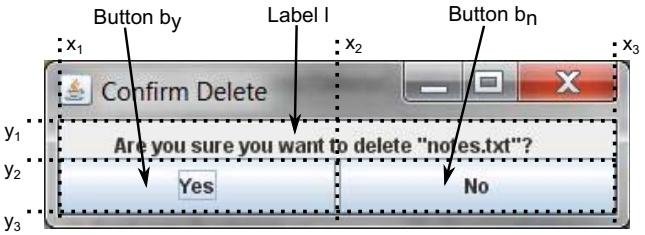
We propose two algorithms for solving systems of prioritized linear constraints with the linear relaxation method. The first algorithm successively adds non-conflicting constraints in descending order of priority. The second algorithm starts with all constraints and successively removes conflicting constraints in ascending order of priority. These algorithms are described in Section V.

The methodology and the results of an evaluation are described in Section VI. The proposed extensions of linear relaxation were evaluated with regard to their convergence and performance, using randomly generated UI layout specifications. The results show that most of the proposed algorithms are efficient, in particular they outperform LP-Solve, a well-known linear programming solver that has been used for UI layout, and QR-decomposition, a direct method. One of our motivations is to develop solvers that have a smaller and more flexible codebase so that they can be more easily used, e.g. in web-based applications. LP-Solve, for example, is not a pure Java program and therefore cannot be used in certain web applications that our user interface technology is targeted at. The evaluation indicates that our best algorithms can be used efficiently in such situations. Our conclusions can be found in Section VIII.

II. User Interface Layout as a Linear Problem

Constraints are a suitable mechanism for specifying the relationships among objects. They are used in the area of logic programming and artificial intelligence, but also for user interfaces. 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 recognized as a powerful method for specifying UI layouts, where the objects are widgets and the relationships between them are spatial relationships such as alignment and proportions [4]. In addition to the relationships to other widgets, each widget has its own set of constraints describing properties such as minimum, maximum and preferred size.

UI layouts are often specified with linear constraints [5]. 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. The resulting systems of linear constraints are sparse. 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 Fig. 1. As a result, the direct interaction between constraints is limited by the topology of a layout, resulting in sparsity.



Label l Hard: $x_3 - x_1 \geq l.minwidth$, $y_2 - y_1 \geq l.minheight$
Soft: $x_3 - x_1 = l.prefwidth$, $y_2 - y_1 = l.prefheight$
Button b_y Hard: $x_2 - x_1 \geq b_y.minwidth$, $y_3 - y_2 \geq b_y.minheight$
Soft: $x_2 - x_1 = b_y.prefwidth$, $y_3 - y_2 = b_y.prefheight$
Button b_n Hard: $x_3 - x_2 \geq b_n.minwidth$, $y_3 - y_2 \geq b_n.minheight$
Soft: $x_3 - x_2 = b_n.prefwidth$, $y_3 - y_2 = b_n.prefheight$

Fig. 1: Example layout with hard and soft constraints

For sparse linear problems, linear relaxation is known to perform very well. However, linear relaxation in its standard form cannot be applied to the UI layout problem for two reasons. First, the coefficient matrices are non-square: there are usually more constraints than variables. Secondly, 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 linear relaxation. Some of these solvers will be discussed in Section V-A.

III. Linear Relaxation

The approximate methods that provide solutions for systems of linear constraints starting from an initial estimate are known as iterative methods. Most of the research on iterative methods deals with iterative methods for solving linear systems of equalities and inequalities for sparse square matrices, the most important method being linear relaxation. This section summarizes the most important findings.

The best known iterative method for solving linear constraints is the Gauss-Seidel algorithm [1]. Given a system of *n* equations and *n* variables of the form

$$Ax = b \tag{1}$$

we can rewrite the equation for the ith term as follows:

$$x_i^{r+1} = \frac{1}{a_{ii}}(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{r+1} - \sum_{j=i+1}^{n} a_{ij}x_j^r) \tag{2}$$

with r is the iteration number and i is the row index. 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. After an initial estimate for x is chosen, it is substituted into the right-hand side of the above equation to produce the next approximation. Iteration is continued until the relative approximate error is less than a pre-specified tolerance.

Linear relaxation, also known as successive over-relaxation (SOR), is an improvement of the Gauss-Seidel method [6]. It is used to speed up the convergence of the Gauss-Seidel method by introducing a parameter w, known as relaxation parameter so that

$$x_i^{r+1} = \frac{w}{a_{ii}}(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{r+1} - \sum_{j=i+1}^{n} a_{ij}x_j^r) + (1-w)x_i^r \tag{3}$$

SOR reduces to the Gauss-Seidel method when $w = 1$. It is known as over-relaxation if $w > 1$, and known as under-relaxation if $w < 1$. One of the main advantages of using linear relaxation is that it is a simple and robust method, which has the ability to handle large sparse problems.

A. Convergence

A frequently used sufficient condition for the convergence of the Gauss-Seidel method is the property of diagonal dominance. There are other characterizations of convergence in the literature, e.g. based on the spectral radius of a normalized matrix, but we focus here on diagonal dominance.

A matrix is said to be diagonally dominant if the absolute value of the diagonal element in each row is greater than or equal to the sum of the absolute values of the rest of the elements in that particular row [7]:

$$|a_{ii}| \geq \sum_{j \neq i} |a_{ij}| \tag{4}$$

for all i. Convergence is guaranteed if the problem matrix is diagonally dominant.

This gives naturally rise to a more general quantity that describes the influence of a variable and that can be used to restate the condition of diagonal dominance.

Definition 1. *The influence of the kth variable in the ith constraint is*

$$\textit{influence}_{ik} = \frac{|a_{ik}|}{\sum_j |a_{ij}|} \tag{5}$$

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. A matrix is diagonally dominant if all diagonal variables have an influence greater or equal to $\frac{1}{2}$. If a coefficient matrix is diagonally dominant and $0 < w < 2$, then linear relaxation is guaranteed to converge [8].

B. Inequalities

Linear relaxation supports linear equalities as well as inequalities. Inequalities are handled similar to equalities [9]–[11]: 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 2. *Mixed System: A system containing equalities as well as inequalities is called a mixed system.*

Definition 3. *Maximum equality subsystem: The subsystem that consists of all the equations in a system of linear constraints is called maximum equality subsystem.*

A mixed system with a square 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 in UI layout, 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 linear relaxation algorithm, which works only on a square matrix, is insufficient.

C. Advantages

Iterative methods such as linear relaxation have certain advantages over direct methods. Iterative algorithms are typically simpler than direct ones, hence can be implemented in smaller programs. Furthermore, they have fewer round-off errors compared to direct methods [3]. 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 [3]. This also means that iterative methods need not store zero-coefficients explicitly, and hence generally use less memory than direct methods. Considering these advantages, it would be useful if the limitations of linear relaxation could be overcome.

IV. Non-Square Matrices

As pointed out in Section III-B, 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 linear relaxation algorithm cannot be applied. In this section, we describe two algorithms that can be used to overcome these limitations.

A. Related Work

Linear systems with non-square matrices are typically solved using direct methods, such as the normal equations [12] and the QR-factorization [7] method.

The normal equations 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 the original matrix is converted into the normal equation matrix and the right-hand side vector, information can be lost. The normal equations method is considered the most common method despite the loss of information because, as shown in [13], an accurate solution for the normal equations can be achieved with iterative refinement [14].

The QR-factorization method is a direct method 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. In contrast to the normal equations method, these methods require the calculation of a significant number of norms, which makes them slower [7].

There are some iterative methods [15] that can be used to solve systems of linear equations that are over-determined. These methods include the simplex [16] and generalized minimal residual [17] method. They have some limitations that make them inapplicable for some problems. These limitations are described as follows.

The simplex algorithm [16] 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 [18] is a variant of the simplex algorithm which is computationally efficient for large sparse problems.

The generalized minimal residual method [17] is considered the most efficient method for solving least squares problems. This method converts a non-square matrix into a square matrix by applying normal equations. One of its shortcomings is instability and poor accuracy of the computed solution due to the possibly high ill-conditioning of the normal equations system. Several methods for iteratively solving linear least squares problems are surveyed in [17]. These methods are also known as Krylov subspace methods.

Numerical difficulties encountered for under-determined problems are the same as in over-determined problems as described above. 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.

B. 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 4. *An assignment of constraints to variables is called pivot assignment.*

$$\gamma\colon \textit{Constraints} \to \textit{Variables}$$

A pivot assignment γ is called feasible *if it is* surjective *and* total.

Surjectiveness is necessary because we require at least one constraint for each variable, otherwise the variable's value would not be changed by the algorithm. The requirement of totality is inherent in the definition of the linear relaxation algorithm, which requires a pivot variable for every constraint.

C. Extension of Linear Relaxation

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

Definition 5. *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.*

D. Random Pivot Assignment

The random algorithm assigns the pivot variable for each constraint randomly in each iteration. This means that in general the pivot assignment is changed for each iteration. To prevent ill-conditioned situations, the algorithm makes sure that variables with influence close to zero are never selected.

It is not inherently obvious that randomized assignments work for the linear relaxation approach, but it is the simplest approach that may work. Although the random algorithm does generally not make the optimum 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 observed drawbacks of random assignment is that it causes more fluctuation of 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.

E. Deterministic Pivot Assignment

The deterministic pivot assignment algorithm is only needed for the equalities, therefore the algorithm described here is applied to the maximum equality subsystem. Choosing a variable in an inequality as a pivot element is not enough to satisfy the criterion "choose every variable at least once" since an inequality will not be further considered if it is fulfilled. The deterministic algorithm for pivot assignment, working on the maximum equality subsystem, is given in Fig. 2. It creates a single pivot assignment that is used consistently during the solving process. It is explained in the course of the following proof of correctness. For the inequalities, one can choose the most influential variable as a pivot variable. However, in general it must be ensured that the same variable is not chosen in two subsequent constraints.

Theorem 1. *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 been assigned a constraint yet 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 correctly chooses the diagonal elements as pivots in the case of diagonally dominant matrices. The algorithm may duplicate constraints. Since this does not change the solution of the problem, this is a valid transformation.

Input: Constraints
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 γ(c) = x
4:    else
5:      Choose the most influential variable x of c, assign
        γ(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 γ(c') = y
11: end for
```

Fig. 2: Deterministic pivot assignment

V. Soft Constraints

Hard constraints are constraints that must always be satisfied. If this is impossible, there is no solution. For many problems, including 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, which 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. 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 $E \subseteq \mathit{Constraints}$ of *enabled constraints*. We consider the characteristic function $\mathbf{1}_E : \mathit{Constraints} \to \{0, 1\}$ of E as an integer in binary representation. 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. In the following, we discuss existing approaches for solving linear soft constraints. Then, we describe two algorithms that address support for soft constraints in the linear relaxation method: constraint insertion and constraint removal.

A. Related Work

All constraint solvers for UI layout must support over-constrained systems. There are two approaches: weighted constraints and constraint hierarchies. Weighted constraints are typically used with direct methods, while constraint hierarchies are used with linear programming.

Direct methods for soft constraints are least squares methods such as LU-decomposition, QR-decomposition [19] and Householder reflections. The UI layout solver HiRise [20] is an example of this category. HiRise2 [21] is an extended version of the HiRise constraint solver which solves hierarchies of linear constraints by applying an LU-decomposition-based simplex method.

Many UI layout solvers are based on linear programming and support soft constraints using slack variables in the objective function [5], [22]–[25].

Input: Constraints
Output: Non-conflicting constraints

```
1:  DISABLE all constraints
2:  SORT constraints 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 linear relaxation
8:     if solution not optimal then
9:        DISABLE c
10:       Restore old variable values
11:    end if
12: end for
```

Fig. 3: Constraint insertion algorithm

Many different local propagation algorithms have been proposed for solving constraint hierarchies in UI layout. The DeltaBlue [26] and SkyBlue [27] algorithms are examples of this category. These algorithms arrange weaker constraints prior to stronger constraints. They cannot handle simultaneous constraints that depend on each other.

B. Constraint Insertion

The first algorithm that we propose for solving a system of soft constraints is called constraint insertion and is defined in Fig. 3. In this algorithm, we test constraints incrementally. We start with an empty set E of enabled constraints (line 1). Iterating through the constraints in order of descending priority, 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, then we proceed to the next constraint. If no solution is found within a fixed maximum number of iterations, then 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 an adequate relaxation parameter is chosen so that linear relaxation converges if there is no conflict. For every feasible linear specification, there is a linear relaxation parameter so that linear relaxation converges [28]. The algorithm is approximating the maximum characteristic function starting from the most significant bit.

C. Constraint Removal

Constraint removal, which is defined in Fig. 4, was used successfully during our evaluation but has a limitation with regard to the set E of constraints that are solved. In contrast to constraint insertion, non-conflicting constraints of lower priority may be lost if there are conflicting constraints of higher priority in a specification. We present this approach to provide another perspective on addressing soft constraints, and as a pointer to future work.

We start with all constraints enabled, i.e. $E = \textit{Constraints}$ (line 1). We try to solve the specification, and assuming that

Input: Constraints
Output: Non-conflicting constraints

```
1:  ENABLE all constraints
2:  SORT constraints by priority
3:  Assign pivot elements for all constraints
4:  for each constraint do
5:     Apply linear relaxation
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:          // c is the conflicting constraint with the lowest
             priority
12:          DISABLE c
13:          Assign pivot elements for all remaining constraints
14:          break
15:       end if
16:    end for
17: end for
```

Fig. 4: Constraint removal algorithm

an adequate linear relaxation parameter has been chosen, a solution is found if 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.

The algorithm as described above would provide correct results, according to the requirement that the integer corresponding to the characteristic function of E is maximized. The problem is the definition of the predicate *conflicting*. Currently, we use a heuristic: we treat a constraint c as conflicting if the value of its pivot variable $\gamma(c)$ has been changed significantly during the last linear relaxation iteration. While this is true for conflicting constraints, this is not a sufficient condition, as other non-conflicting constraints may be affected by a conflict and hence satisfy this condition, too. The consequence is that some constraints with a priority below that of the conflicting constraint with the lowest priority may be removed, too.

VI. Evaluation

In this section we present an experimental evaluation of the proposed algorithms with regard to their convergence, and their performance.

A. Methodology

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.

Layout specifications were randomly generated using the test data generator described in [5]. For each experiment the same set of test data was used. The specification size was varied from 6 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

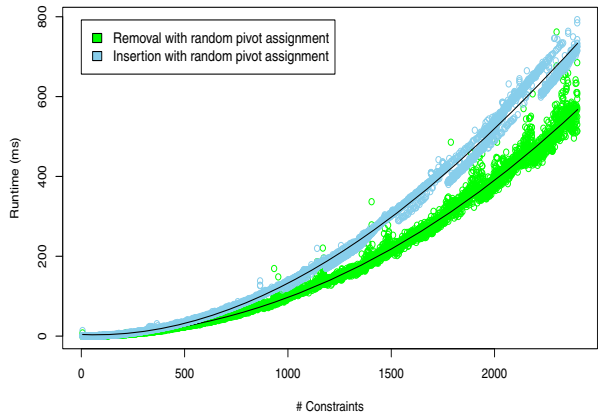


Fig. 5: Performance comparison between constraint insertion and removal using random pivot assignment

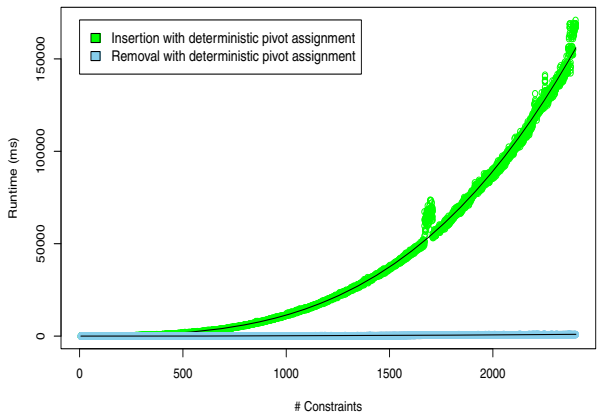


Fig. 6: Performance comparison between constraint insertion and removal using deterministic pivot assignment

layouts were generated resulting in a total of 6000 different layout specifications which were evaluated. A linear relaxation parameter of 0.7 and a tolerance of 0.01 were used for linear relaxation. As a reference, all the generated specifications were also solved with LP-Solve [29], which is a well-known efficient linear programming solver, and QR-decomposition, using the Apache Commons Mathematics Library [30]. LP-Solve is written in C and compiled to native code, which gives it a slight performance advantage compared to our Java implementations.

Solving of each of the random layout specifications was repeated 10 times. In the performance graphs, individual measurements are illustrated as differently shaded circles. We applied several types of regression models (linear, quadratic, log, cubic) to the performance data. The polynomial model $(x + x^2 + x^3)$ showed the best fit for the performance data of all solvers ($R^2 > .92$). The fitted polynomials are shown as black lines in the graphs.

B. Results

The performance results are shown in Figs. 5, 6 and 7. Figure 5 illustrates the performance comparison of constraint insertion and constraint removal using random pivot assignment. Constraint removal exhibited a better performance than constraint insertion.

Figure 6 compares constraint insertion and removal using deterministic pivot assignment. Insertion using deterministic assignment is slow because after adding each of the constraints the pivot assignment has to be recomputed. In comparison, the runtime of constraint removal using deterministic pivot assignment appears almost linear in the number of constraints. One reason for this difference is that for removal the pivot assignment needs only be recomputed for each conflicting constraint.

Figure 7 compares all the aforementioned algorithms except the slow insertion with deterministic pivot assignment

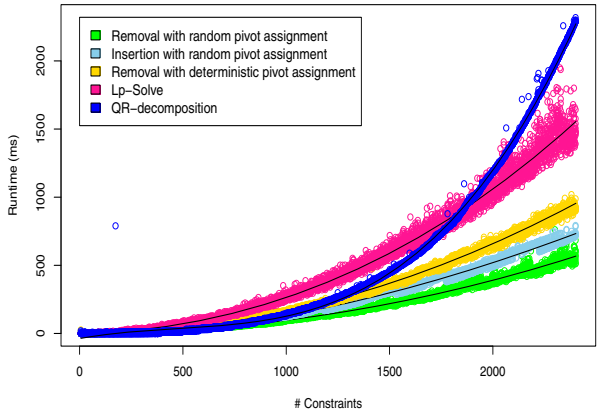


Fig. 7: Performance comparison of all algorithms with LP-Solve and QR-decomposition

to LP-Solve and QR-decomposition. All our algorithms in Figure 7 performed significantly better than LP-Solve and QR-decomposition. The convergence results of our proposed algorithms were all optimal.

VII. Discussion

The performance results showed that constraint insertion with deterministic pivot assignment does not have a satisfactory performance compared to our other algorithms. All our proposed algorithms, except insertion with deterministic pivot assignment, are faster than an implementation of QR-decomposition and the simplex algorithm implementation of LP-Solve. As described earlier, direct methods suffer from fill-in effect when solving sparse systems, which generally makes them inferior to indirect, iterative methods in this case.

We already mentioned that constraint removal suffers from the limitation that sometimes non-conflicting constraints are removed in the presence of conflicting constraints. However, using constraint insertion with random pivot assignment does not have this limitation and performs almost as well as constraint removal. Hence, constraint insertion with random pivot assignment appears to be the most appropriate algorithm.

One of the limitations of using indirect, iterative methods is that convergence is not guaranteed in certain circumstances. They may not converge if the conditions described in Section III are not fulfilled. However, the two proposed algorithms converged for all the tested 6000 specifications, yielding (with the aforementioned exception of constraint removal) optimal results. This is a strong indication that in practice, convergence is not a problem for UI layout specifications.

VIII. Conclusion

We have proposed new algorithms for using linear relaxation for solving constraint-based UI layout problems. In particular, we presented the following contributions:

- Algorithms for pivot assignment that make it possible to solve non-square matrices.
- Support for soft constraints that makes it possible to solve over-constrained specifications.
- An evaluation that indicates that most of the proposed algorithms are optimal and outperform a modern linear programming solver, LP-Solve, and a QR-decomposition solver.

These algorithms, in particular constraint insertion with random pivot assignment, bring the benefits of efficiently solving sparse matrices to UI layout.

As a future work, a more accurate way of detecting conflicting constraints for the constraint removal algorithm needs to be found. Another possible future direction is to extend the proposed algorithms to solving of non-linear UI constraint problems.

References

[1] A. B. Saeed and A. B. Naeem, *Numerical Analysis*. Shahryar, 2008.

[2] S. Kunis and H. Rauhut, "Random sampling of sparse trigonometric polynomials, ii. orthogonal matching pursuit versus basis pursuit," *Journal Foundations of Computational Mathematics*, vol. 8, no. 6, pp. 737–763, Nov. 2008.

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

[4] C. Zeidler, J. Müller, C. Lutteroth, and G. Weber, "Comparing the usability of grid-bag and constraint-based layouts," in *OzCHI '12: Proceedings of the 24th Australian Computer-Human Interaction Conference*. New York, NY, USA: ACM, 2012.

[5] C. Lutteroth, R. Strandh, and G. Weber, "Domain specific High-Level constraints for user interface layout," *Constraints*, vol. 13, no. 3, 2008.

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

[7] B. N. Datta, *Numerical Linear Algebra And Applications*. Cole, 1995.

[8] C. G. Broyden, "On convergence criteria for the method of successive over-relaxation," *Mathematics of Computation*, vol. 18, pp. 136–141, 1964.

[9] J. L. Goffin, "The relaxation method for solving systems of linear inequalities," *Mathematics of Operations Research*, vol. 5, no. 3, pp. 388–414, 1980.

[10] S. Agmon, "The relaxation method for linear inequalities," *Canadian Journal of Mathematics*, pp. 382–392, 1954.

[11] T. Motzkin and I. Schoenberg, "The relaxation method for linear inequalities," *Canadian Journal of Mathematics*, pp. 393–404, 1954.

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

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

[14] A. Bjork, "Error analysis of least squares problems," *Proc.of the NATO Adv.Study Inst. On Numerical Linear Algebra,Digital Signal Processing and Parallel Algorithms*, 1988.

[15] X. Wang, "Incomplete factorization preconditioning for linear least squares problems," *Technical Report, University of Illinois,USA*, pp. 1–212, 1984.

[16] G. B. Danzig, *Linear Programming and Extensions*, 11st ed., ser. Princeton Landmarks in Mathematics. Princeton Uni. Press, 1998.

[17] G. Golub and C. Van Loan, *Matrix Computations*. Johns Hopkins Uni. Press, 1996.

[18] H. A. Taha, *Operations Research: An Introduction*. Mcmillan, 1992.

[19] Y. Yoshioka, H. Masuda, and Y. Furukawa, "A constrained least squares approach to interactive mesh deformation," in *Proceedings of the IEEE International Conference on Shape Modeling and Applications 2006*, ser. SMI '06. IEEE Computer Society, 2006, pp. 23–.

[20] H. Hosobe, "A scalable linear constraint solver for user interface construction," in *Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming*, ser. CP '02. Springer, 2000, pp. 218–232.

[21] ——, "A simplex-based scalable linear constraint solver for user interface applications," in *Tools with Artificial Intelligence (ICTAI), 2011 23rd IEEE International Conference on*, Nov. 2011, pp. 793–798.

[22] G. J. Badros, A. Borning, and P. J. Stuckey, "The cassowary linear arithmetic constraint solving algorithm," *ACM Transactions on Computer-Human Interaction*, vol. 8, no. 4, pp. 267–306, 2001.

[23] A. Borning, K. Marriott, P. Stuckey, and Y. Xiao, "Solving linear arithmetic constraints for user interface applications," in *Proceedings of the 10th annual ACM symposium on User interface software and technology (UIST '97)*. ACM, 1997, pp. 87–96.

[24] M. S. Bazaraa, J. J. Jarvis, and H. D. Sherali, *Linear Programming and Network Flows*, 4th ed. Wiley, 2009.

[25] K. Marriott, S. C. Chok, and A. 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*, ser. CP '98. Springer, 1998, pp. 340–354.

[26] J. M. Freeman-Benson and A. Borning, "An incremental constraint solver," *Communications of the ACM*, vol. 33, no. 1, pp. 54–63, 1990.

[27] M. 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 (UIST '94)*. ACM, 1994, pp. 137–146.

[28] R. L. Burden and J. Faires, *Numerical Analysis*. Bob Pirtle, 2005.

[29] M. Berkelaar, J. Dirks, K. Eikland, P. Notebaert, and J. Ebert. (2012) Lp-solve: A (mixed integer) linear programming problem solver. [Online]. Available: http://lpsolve.sourceforge.net/

[30] Apache Software Foundation. (2012) Commons Math: The Apache Commons mathematics library. [Online]. Available: http://commons.apache.org/math