Constraint Solvers for User Interface Layout

Noreen Jamil
Department of Computer Science
University of Auckland
Private Bag 92019
Auckland, New Zealand
fnjam031g@aucklanduni.ac.nz

ABSTRACT: *Constraints have played an important role in the construction of GUIs, where they are mainly used to define the layout of the widgets. Resizing behavior is very important in GUIs because areas have domain specific parameters such as form the resizing of windows. If linear objective function is used and window is resized then error is not distributed equally. To distribute the error equally, a quadratic objective function is introduced. Different algorithms are widely used for solving linear constraints and quadratic problems in a variety of different scientific areas. The linear relxation, Kaczmarz, direct and linear programming methods are common methods for solving linear constraints for GUI layout. The interior point and active set methods are most commonly used techniques to solve quadratic programming problems. Current constraint solvers designed for GUI layout do not use interior point methods for solving a quadratic objective function subject to linear equality and inequality constraints. In this paper, performance aspects and the convergence speed of interior point and active set methods are compared along with one most commonly used linear programming method when they are implemented for graphical user interface layout. The performance and convergence of the proposed algorithms are evaluated empirically using randomly generated UI layout specifications of various sizes. The results show that the interior point algorithms perform significantly better than the Simplex method and QOCA-solver, which uses the active set method implementation for solving quadratic optimization.*

Keywords: UI Layout, Interior Point, Simplex Method, Quadratic Problems

Received: 27 May 2013, Revised 3 July 2013, Accepted 12 July 2013

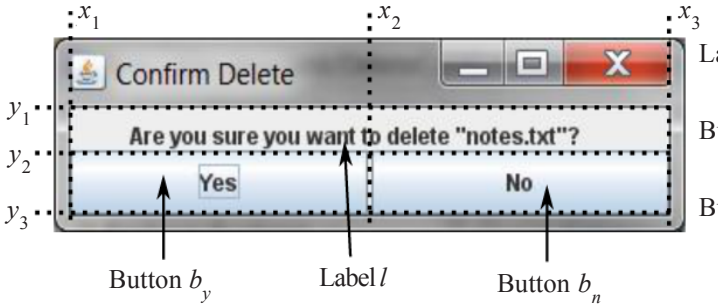
1. Introduction

Constraints are a suitable mechanism for specifying the relationships among objects. They are used in the area of logic programming, artificial intelligence and UI specification. 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 common way of specifying UI layouts, where the objects are widgets and the relationships between them are spatial relationships such as alignment and proportions. 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 [1]. 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. Furthermore, 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 Figure 1. As a result, the direct interaction between constraints is limited by the topology of a layout, resulting in sparsity.

The Auckland Layout Model (ALM) [2] enables the description of Graphical User Interfaces (GUIs) in a constraintbased manner. Instead of placing widgets with absolute or relative coordinates on a window the relations between them are specified with constraints a GUI has to fulfill [3]. Therefore it is easier to realize highly adaptable GUIs and achieve a better modularity of GUI elements compared with common GUI techniques such as Javas Gridbag Layout.



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: $x_3 - x_1 = b_y$.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

Figure 1. Example constraint-based UI layout with hard and soft constraints

Constraints are the way to formulate requirements on a specific GUI in ALM. A complete layout in ALM is therefore defined by a set of areas (defined by a set of tab stops) and a set of constraints. With the given set of constraints an ALM layout manager has to solve basically a system of linear equalities and inequalities. However it is often the case in layout definitions that the system is over-specified. To cope with that problem ALM introduces the notion of soft constraints [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. To solve layouts which are defined with soft constraints and inequalities it is not sufficient to solve a system of linear equations but it is required to introduce a sort of optimization, namely the minimization of the constraint-violation. The violation is modeled with an additionally introduced penalty parameter for each softconstraint.

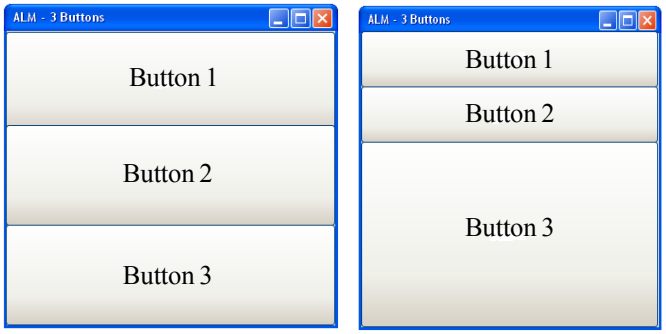
Current implementations of ALM use the simplex algorithm for that task. However, with more complex GUI specifications the responsiveness of the GUI decreases due to an increasing computational effort. To increase the computational speed a linear relaxation algorithm [4] is currently used with a linear objective function. One of the drawbacks of using linear objective function is the violation of soft constraints in a nonuniform way as shown in Figure 2, where only few constraints are violated but it is not precise which constraints are violated. This leads to the development of a quadratic objective function which minimizes the square of the deviations from a solution point to the defined constraints of a system.

Several different constraint-based GUI layout technologies have been developed and each of these technologies has their own peculiarities and requires specific knowledge. By using these programmers and even end users can easily solve their problems with constraints since they have only to describe the problems. The constraint idea was originated by Sutherland in 1960 who introduced Sketchpad [5], the first interactive graphical interface that solved geometric constraints. Since then many constraint solvers have been developed and studied by the research community [1], [6], [7] and interest has increased with the recently introduced constraint-based layout model in the Cocoa API of Apple's Mac OS X[1]. Several different constraint-based GUI layout technologies have been developed and each of these technologies has their own peculiarities and requires specific knowledge. By using these programmers and even end users can easily solve their problems with constraints since they have only to describe the problems. The constraint idea was originated by Sutherland in 1960 who introduced Sketchpad [5], the first interactive graphical interface that solved geometric constraints.

Most researchers have concentrated on developing and improving the performance of general algorithms for the solution of many complex problems. This is due to the rapid increase in the advancement in computer hardware (high speed processors,

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

large memory etc). While Cassowary [8] was one of the pioneers in developing algorithms for fast solution of GUI layout problems. One of the challenges of the last few decades has been the construction of fast numerical solution algorithms for solving GUI layout problems.



(a) Three buttons with quadratic objective function

(b) Three buttons with linear objective function

Figure 2. Two different solving strategies for a simple two-button layout

Recent developments in iterative methods have improved the efficiency of these methods. The use of iterative methods has become ubiquitous in recent years for solving sparse, realworld optimization problems where direct algorithms are not suitable due to fill-in effects. Unlike direct algorithms, which try to solve the problems finitely, iterative methods start with a complete but preliminary task that is not necessarily consistent. They improve this task in several iterative steps until specified stopping criteria are satisfied. We can get good approximate solutions by iterating the process, which is useful for practical applications especially if an efficient solution is required.

Much research has been carried out on constraint solving techniques for linear programming problems. However, the linear programming technique tends to use an iterative method along with one step of a direct method. Therefore, it is worth studying the potential of iterative algorithms because of their efficiency and capability to solve sparse linear constraint problems.

In this paper, we compare constraint solving techniques which are using quadratic and linear objective function. These techniques were experimentally evaluated with regard to convergence and performance, using randomly generated UI layout specifications. The results show that the interior point method is more efficient than the active set and simplex methods. The simplex and active set methods have previously been used for UI layout. In section VI, we discuss related work. A detailed description of quadratic programming and how systems of layout constraints can be solved using a quadratic objective functions is given in Section III. Linear programming description and some overview of simplex algorithm is described in detail in Section IV. The methodology as well as the results of the evaluation can be found in Section V. Section VI finishes with conclusions and an outlook on future work.

2. Related Work

Most of the research related to GUI layout deals with the various algorithms for solving constraint hierarchies. Research related to constraint based UI layout has provided results in the form of tools [9], [10] and algorithms [7], [8] for specific tasks. The latest work [11] on constraint based GUIs uses a quadratic solving strategy which they find better than linear solving strategies. They [11] implemented the active set method for solving a quadratic objective function subject to some linear constraints. Baraf [12] presents a quadratic optimization algorithm for solving linear constraints in modelling physical systems. QOCA [7] uses the active set algorithm for solving quadratic programming problem for graphical user interface layout.

All constraint solvers for UI layout have to support overconstrained 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. Examples of direct methods for soft constraints are HiRise and HiRise2 [9]. Many UI layout solvers are based on linear programming and support soft constraints using slack variables in the objective function [1], [7], [8].

Many different local propagation algorithms have been proposed for solving constraint hierarchies in UI layout. The DeltaBlue

[13], SkyBlue [14] and Detail [15] algorithms are examples in 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 to handle soft constraints utilized in these solvers are designed to work with direct methods, so they inherit the problems direct methods usually have with sparse matrices.

None of the above discussed algorithms apply interior point methods for UI layout.

3. Quadratic Programming

Quadratic programming deals with the optimization (minimization or maximization) of quadratic objective function that satisfies set of linear constraints.

Definition 1. *Quadratic Programming is a problem which can be formulated as*:

$$q(\bar{x}) := \frac{1}{2}\,\bar{x}^T Q\bar{x} - \bar{g}^T\bar{x} \rightarrow min$$

Subject to $Ax = b$ *and* $Cx \leq b$, $x \geq 0$, *where the Hessian matrix* Q (*positively semi-definite*) *of the quadratic function is an n* $\times$ *n quadratic matrix and* x^T *is a vector transpose of x.*

$Ax = b$ *and* $Cx \leq b$ is a set of linear equality and inequality constraints where $x \geq 0$ requires non-negativity conditions. Generally, two approaches are used to solve quadratic programming problems. These approaches are: interior point and active set. The active set method is preferable if the QP problem is medium but the matrices are dense. As large and sparse problems occur in user interface (UI) layout the method of choice is an interior point which is described in detail in the following section.

3.1 Interior Point Method

We use an interior point algorithm, the barrier method [16] for constraint based Graphical User Interfaces, which realizes the error distribution with a quadratic objective function. It is an iterative method for solving constrained optimization problems with inequality constraints. It is reasonably efficient and scalable up to thousand of constraints. An interior point algorithm is one that starts inside the feasible area and reaches the optimal solution through the interior of the feasible region.

The key idea behind the algorithm is to start with a feasible point and a relatively large value of the parameter *r* (where *r* is the barrier parameter). A "*barrier*" function is used in order to define the feasible region of the domain, i.e. to satisfy inequality constraints. As we proceed with the iteration, the barrier function becomes steeper, so it forces to be into the feasible region. Note that the feasible region is in both case an inner region (that's where the word "*interior*" comes from), that is the inequality constraints are always strict constraints (<; not <=). The barrier method uses an outer iteration in which the barrier gets raised, and an inner iteration that solves this i th particular problem $i - th$, and so on until the final convergence. The barrier term is added to the objective function for a maximization problem and subtracted for a minimization problem. The details of algorithm are as follows.

• **Step 1:** Initialization Step: First choose an initial barrier parameter $m > 1$, then a stopping parameter $\epsilon > 0$, and a strictly feasible x that violates at least one constraint and formulate the augmented objective function.

• **Step 2:** Repeat:

• **Step 3:** Centering Step: Computing $x_o(t)$ by minimizing $tf_o + \phi$ subject to $Ax = b$, starting at x.

• **Step 4:** Updating Step: Update. $x := x_o(t)$.

• **Step 4:** Stopping Rule: Quit if $m / t < \epsilon$ Increase t. $t := \mu t$.

Here f_o is the objective function, ϕ is the barrier function for the given problem and t is the weight of the barrier function in the i-*th* sub problem. As t increases, the barrier function becomes steeper and better approximates the inequality constraints. The starting point must be strictly feasible for all of the constraints.

At each iteration the algorithm computes the central point $x_o(t)$ from the previously computed central point, and then increases t by a factor of $\mu > 1$. At the end we terminate the algorithm if $m / t < \epsilon$, otherwise we repeat the process.

Our solver uses the Gurobi library [17] for implementation of the interior point algorithm to solve our convex problem. It requires Gurobi to be installed and the Gurobi jar library in the build path.

3.2 Active Set Method

The active set method [18] is an iterative method for solving quadratic programming problems. The idea behind the active set method is to solve a sequence of quadratic programming problem. Each problem consists of a set of an objective function subject to equality constraints, which is known as the active set. An active set contains equality constraints as well as inequality constraints (which are not active but must be fulfilled as equalities). The active set method solves the quadratic programming problem by identifying the active set of its solution. This method solves the hard constraints of the form:

$$A_i \,.\, \bar{x} = \bar{b}_i \qquad i \,\epsilon\, equalities$$
$$A_i \,.\, \bar{x} \,^3\, \bar{b}_i \qquad i \,\epsilon\, equalities$$

while minimizing the objective function:

$$q(\bar{x}) := \frac{1}{2}\, \bar{x}^T\, Q\bar{x} -\ \bar{g}^T \bar{x} \rightarrow min$$

where Q is symmetric. The equality problem

$$A.\ \bar{x} = \bar{b}$$

$$q(\bar{x}) := \frac{1}{2}\, \bar{x}^T\, Q\bar{x} -\ \bar{g}^T \bar{x} \rightarrow min$$

is analytically solvable by setting

$$\nabla q(\bar{x}) := 0$$

and solving the linear problem. Steps for the active set algorithm are as follows:

Step 1: Find a base solution for the hard constraints (linear system of inequalities)

$$A.\ \bar{x} \geq \bar{b}$$

Step 2: Create an initial active set A holding all soft constraints which satisfy the base solution as **equality** constraints.

Step 3: From the base solution $\bar{x}$ find a new $\bar{x}_{new} = \bar{x} + \bar{\delta}$ which optimizes a objective function of the active set A (soft constraints).

This implies for the hard constraints:

$$A\bar{x}_{new} = \bar{b}$$

$$\Leftrightarrow A\bar{x} + A\bar{\delta} = \bar{b}$$

$$\Rightarrow A\bar{\delta} = 0$$

Get δ $(\bar{x}_{new} = \bar{x} + \bar{\delta})$

δ should lead us nearer to the optimal solution in respect to

$$q(\bar{x}) := \frac{1}{2}\, \bar{x}^T\, Q\bar{x} -\ \bar{g}^T \bar{x} \rightarrow min$$

$\Rightarrow$ let δ point to the searched minimum, means $\delta \rightarrow \nabla q(x^k)$

Get δ $(\bar{x}_{new} = \bar{x} + \bar{\delta})$

This leads to the new quadratic sub problem:

$$k(\bar{x}) := \frac{1}{2}\bar{\delta}^T\, Q\bar{\delta} - \nabla q(x^k)^T \bar{\delta} \rightarrow min$$

subject to

$$A\bar{\delta} = 0$$

This means δ points as closely as possible in the direction of the derivation of q.

(this is solvable because it is an equality problem)

There are two cases in solving the problem.

Case 1: $\delta = 0$ (remove a constraint) If $\delta = 0$ then remove a constraint that prevents further optimizing of solution: This is the constraint with negative

$$\min \lambda_i = \nabla q\,(x^k)_i$$

If there is no such a constraint (all $\lambda_i >= 0$) then stop.

Case 2: $\delta \neq 0$ (add a constraint) calculate alpha such that:

$$\alpha^k = min\ (1,\ min\ \frac{b_i - (A\bar{x}^k)_i}{(A\bar{\delta}^k)_i})$$

If $\alpha < 1$ it means that the algorithm has not yet reached the constraint edge b_i and if $\alpha = 1$ it means that the algorithm hits the constraint edge

If $\alpha < 1$ then add a constraint with smallest (the constraint that the algorithm hits first) Terminate algorithm if $\bar{x}^{k+1} = \bar{x}^k + \alpha^k . \bar{\delta}$, otherwise repeat the process.

3.2.1 Advantages of Interior Point and Active Set Methods

The interior point method has certain advantages over the active set method. The interior point method is simple to implement and efficient. It is efficient especially 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 [19].

For general nonlinear optimization problems, barrier methods are among the most powerful classes of algorithms [20]. The statement that supports this fact is that these methods will converge to at least a local minimum in most cases, even if the constraints and objective functions do not have convexity characteristics. They work well even in the presence of spinode and similar form that can mystify other approaches.

An interior point is less sensitive to problem size whereas active set adds a combinatorial element to the identification of the active set and the solution of the quadratic programming, and as a result computational effort can increase with the problem size [21].

Considering these advantages, we choose the interior point method for solving GUI layout problems.

4. Linear Programming

Linear programming [22] deals with the optimization (minimization or maximization) of an objective function that satisfies a set of constraints.

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 2. *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.

4.1 Simplex Method

The simplex method [23] 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.

5. Experimental Evaluation

In this section we present an experimental evaluation of the proposed algorithms. We conducted two different experiments to evaluate (i) the convergence behavior, (ii) the performance in terms of computation time. The experiments were conducted as follows.

5.1 Methodology

For both 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 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 [1]. 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 which were evaluated.

This test data served as input for our algorithms. We conducted two experiments. In the first experiment we investigated the convergence behavior 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 right hand- and left hand sides) is not smaller than a given tolerance.

In the second experiment we measured the performance in terms of computational time (T) in milliseconds (ms), depending on the problem size measured in the 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, linear and quadratic constraint solving methods, QOCA solver, the interior point and Simplex methods. For comparison purposes we selected QOCA solver that has been implemented for solving convex quadratic programming problem for UI layout.

The QOCA solver uses largely distinct weights (e.g., 1, 1000, and 1000000) to handle constraint hierarchies. Whereas, we handle soft constraints in our problem formulation by introducing a slack variable per constraint. In the objective function the values of these slack variables are squared and weighted by the penalties of the corresponding constraint.

5.2 Results

In the first experiments we investigated the convergence behavior of all algorithms. We found that both algorithms converge in the end. This result is obvious since the algorithms are designed to find a solvable subproblem.

In the second experiment we investigated the computational time behavior of all algorithms. To figure out 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

$$T = \beta_0 + \beta_1 c + \beta_2 c_2 + \beta_3 c_3 + \epsilon$$

which gave us a good fit for the performance data. Key parameters of the models are shown in Table 2; a graphical representation of the models can be found in Figures 3. Table 1 explains the symbols used.

Symbol	Explanation
β_0	Intercept of the regression model
$\beta_1 - 3$	Estimated model parameters
c	Number of constraints
T	Measured time in milliseconds
R^2	Coefficient of determination of the estimated regression models

Table 1. Symbol Table

For some strategies some parameters do not have a significant effect. That can be interpreted as the complexity of the algorithm not following a certain polynomial trend. As the graphs indicate, interior point exhibits better performance than active set simplex algorithms.

Figure 3 compares interior point, active set and simplex algorithms. Generally, the active set method is slower than the interior point method for solving convex quadratic programming problem for UI layout because of changes in an active set estimate combinatorially. The reason behind slowness of simplex algorithm is one Gauss-Jordan elimination step per iteration, i.e. using a direct method.

Strategy	β_0	β_1	β_2	β_3	R^2
Interior Point Algorithm	$1.179e^{+01***}$	$6.645e^{-03***}$	$1.443e-06^{***}$	$-4.868e^{-10***}$	0.4145
Active Set Algorithm	$1.225e^{+00***}$	$-3.273e^{-03***}$	$9.595e^{-05***}$	$-2.765e^{-09***}$	0.9971
Simplex Algorithm	$-2.491***$	$3.924\ 10^{-02***}$	$2.079.10^{-04***}$	$1.904\ .10^{-08***}$	0.9900

Significance codes: *** $p < 0.001$

Table 2. Regression models for the different solving strategies

5.3 Discussion

The performance results show that the interior point method is faster than the active set method. One reason why the active set method is slow is that it adds a combinatorial element in identifying the active set and the solution of the quadratic programming, and this can increase computational effort for solving the problem. Whereas the interior point method is independent to the problem size.

Even though active set methods have the advantage of warm starts but this approach can be inefficient as explained below. Most of the complex operations (adding and deleting a constraint to the active set, estimates for Lagrange multiplier, computing search directions etc) involved in the make up of algorithm. On the other hand the most complex operation in the interior point method is the solution of a linear system and this operation is fairly simpler than the active set method.

An active set takes large number of iterations to converge for large sparse problems whereas an interior point takes less.

A plausible reason the simplex method is slower is that even though it is an iterative method but it uses one direct method solving step per iteration. As direct methods suffer from fillin effects when solving sparse systems, which is generally a disadvantage compared to iterative methods in this case.

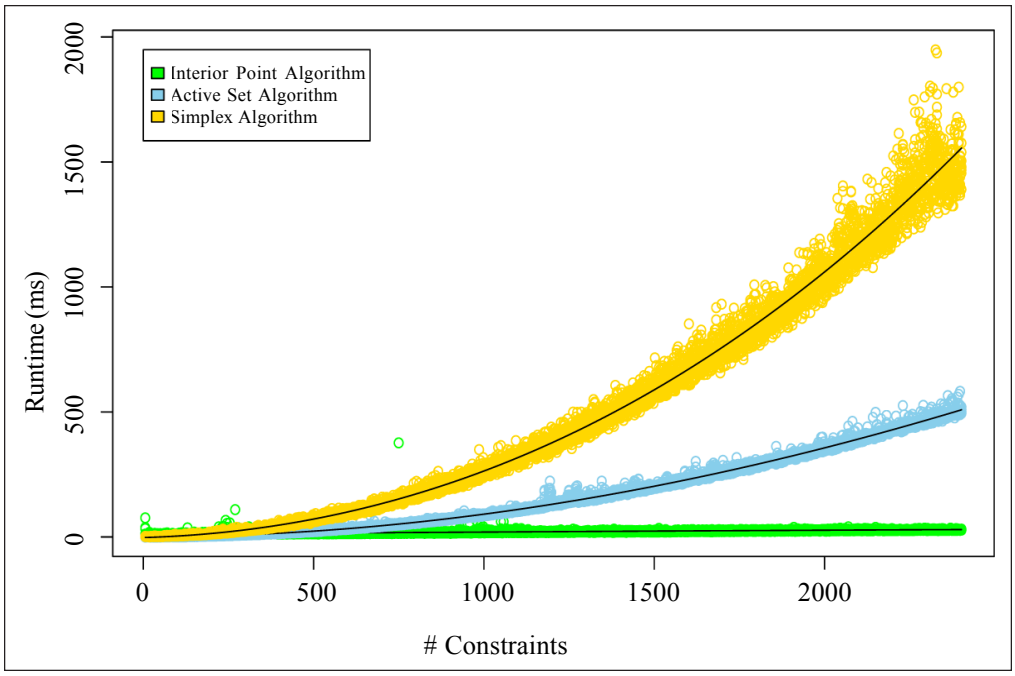


Figure 3. Performance comparison of Interior Point, Active Set and Simplex Algorithms

6. Conclusion

In this paper we have compared the performance of interior point and active set methods for solving convex programming problems for constraint based GUI layouts. We have compared the speed and convergence of these methods. We found that the interior point method is more efficient for large sparse problems than the active set method. We also compared the performance of simplex algorithm and found that it is slowest than interior point and active set methods. The work presented in this paper lays a foundation for the application of iterative methods for solvers of constraint-based UIs. We identify the following future work in that area.

First, some applications in constraint-based UIs could benefit from the possibility to formulate 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.

Second, there is room for improvement in the deterministic pivot assignment algorithm in linear relaxation. The results of the experiment indicate that an optimal pivot assignment can have a huge 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 selector 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.

Third, we have proven the convergence theorem for the Gauss-Seidel method for the case of non-square, row-dominant coefficient matrices. However, our experimental evaluation indicates that linear relaxation converges for some UI lay out problems which do not fully satisfy the row-dominance criterion. A weaker convergence criterion would be very insightful and could lay the basis for further improvement of the algorithms.

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

References

[1] Lutteroth, C., Strandh, R., Weber, G. (2008). Domain specific high-level constraints for user interface layout, Constraints, 13 (3).

[2] Lutteroth, C., Weber, G. (2008). Modular specification of gui layout using constraints, *In*: Proceedings of the 19th Australian Conference on Software Engineering, p. 300–309.

[3] ——. (2006). User interface layout with ordinal and linear constraints, *In*: Proceedings of the 7th Australasian User interface conference, p. 53–60.

[4] Jamil, N., M¨uller, N., Lutteroth, C., Weber, G. (2012). Extending linear relaxation for user interface layout, *In*: Proceedings of 24th International Conference on Tools with Artificial Intelligence (ICTAI), p.1–8, 2012.

[5] Sutherland, I. (1963). Sketchpad: a man-machine graphical communication system, *In*: Proc. IFIP Spring Joint Computer Conference.

[6] Hosobe, H. (2000). A scalable linear constraint solver for user interface construction, *In*: Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming. London, UK: Springer, p. 218–232.

[7] Borning, A., Marriott, K., Stuckey, P., Xiao, Y. (1997). Solving linear arithmetic constraints for user interface applications, *In*: Proceedings of the 10th annual ACM symposium on User interface software and technology (UIST '97). New York, NY, USA: ACM, 1997, p. 87–96. [Online]. Available: http://doi.acm.org/10.1145/263407.263518.

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

[9] Hosobe, H. (2011). A simplex-based scalable linear constraint solver for user interface applications, *In*: Tools with Artificial Intelligence (ICTAI), 23rd IEEE International Conference on, Nov. p. 793–798.

[10] ——. (2000). 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. London, UK: Springer, p. 218–232.

[11] Clemens Zeidler, G. W., Christof Lutteroth. (2012). Constraint solving for beautiful user interfaces: How solving strategies support layout aesthetics, *In*: Proceedings of CHINZ, p. 23–32.

[12] Baraff, D. (1994). Fast contact force computation for nonpenetrating rigid bodies, *Computer Graphics Proceedings*, p. 23–32.

[13] Freeman-Benson, J. M., Borning, A. (1990). An incremental constraint solver, Communications of the ACM, 33 (1) 54–63.

[14] Sannella, M. 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). New York, NY, USA: ACM, p. 137–146. [Online]. Available: http://doi.acm.org/10.1145/192426.192485.

[15] Hosobe, H., Miyashita, K., Takahashi, S., Matsuoka, S., Yonezawa, A. *In*: Proceedings of the Second International Workshop on Principles and Practice of Constraint Programming, London, UK, p. 51–62.

[16] Caroll, W. (1961). The created response surface technique for optimizing nonlinear restrained systems, *Operations Res*, p. 169–184.

[17] Inc, G. O. Gurobi optimizer.software:http://www.gurobi.com/welcome.html.

[18] Fletcher, R. (1987). Practical methods of optimization; (2nd ed.). Wiley- Interscience.

[19] Boyd, S., Vandenberghe, L. (2009). Convex optimization.

[20] Gould, N., Orban, D., Toint, P. Numerical methodsfor large scale nonlinear optimization, ActaNumerica, p. 299–361, 2005.

[21] Ponceleon, D. B. (1991). Barrier methods for large scale quadratic programming, PhD thesis, Department of Computer Science, Stanford University, Stanford, California, USA.

[22] Bazaraa, M. S., Jarvis, J. J., Sherali, H. D. (1990). Linear programming and network flows.

[24] Berkelaar, M., Notebaert, P., Eikland, K. (2007). A (mixed integer) linear programming problem solver: http://lpsolve.sourceforge.net/.

[23] Dantzig, G. B. (1998). Linear programming and extensions, 11st ed., ser. Princeton Landmarks in Mathematics. *Princeton NJ*, USA: Princeton Uni. Press.