, , 1–22 ()
© Kluwer Academic Publishers, Boston. Manufactured in The Netherlands.

Fourier Elimination for Compiling Constraint Hierarchies

WARWICK HARVEY warwick@cs.mu.oz.au
Department of Computer Science and Software Engineering, University of Melbourne, Parkville 3052, Australia[1]

PETER J. STUCKEY pjs@cs.mu.oz.au
Department of Computer Science and Software Engineering, University of Melbourne, Parkville 3052, Australia

ALAN BORNING borning@cs.washington.edu
Department of Computer Science & Engineering, University of Washington, Box 352350, Seattle, Washington 98195, USA

Editor:

Abstract. Linear equality and inequality constraints arise naturally in specifying many aspects of user interfaces, such as requiring that one window be to the left of another, requiring that a pane occupy the leftmost 1/3 of a window, or preferring that an object be contained within a rectangle if possible. For interactive use, we need to solve similar constraint satisfaction problems repeatedly for each screen refresh, with each successive problem differing from the previous one only in the position of an input device and the previous state of the system. We present an algorithm for solving such systems of constraints using projection. The solution is compiled into very efficient, constraint-free code, which is parameterized by the new inputs. Producing straight-line, constraint-free code of this sort is important in a number of applications: for example, to provide predictable performance in real-time systems, to allow companies to ship products without including a runtime constraint solver, to compile Java applets that can be downloaded and run remotely (again without having to include a runtime solver), or for applications where runtime efficiency is particularly important. Even for less time-critical user interface applications, the smooth performance of the resulting code is more pleasing than that of code produced using other current techniques.

Keywords: linear arithmetic constraints, constraint compilation, heirarchical constraints, Fourier elimination algorithm, user interfaces

1. Introduction

Constraints are a natural tool for user interface toolkits and other kinds of interactive graphical systems. Some important uses in this application area include specifying layout and other geometric information, maintaining consistency between application data and a view of that data, and maintaining consistency among multiple views. It is important to be able to express preferences as well as requirements in interactive constraint systems. One important use is to express a desire for stability when moving parts of an image: things should stay where they were unless there is some reason for them to move. A second use is to process potentially invalid user inputs in a graceful way. For example, if the user tries to move a figure outside of its bounding window, it is reasonable for the figure just to bump up against the side of the window and stop, rather than signalling an error. A third use is to balance conflicting desires, for example in laying out a graph. The constraints needed to specify and maintain layout information are typically linear equalities and inequalities over the real numbers.[2] Inequality constraints in particular are needed to express relationships such as "inside," "above," "left-of," and "overlaps." For example, we can express the requirement that `window1` be to the left of `window2` as the constraint `window1.rightSide` $\leq$ `window2.leftSide`. Some of these layout constraints will be requirements, and others preferences.

For interactive systems, a typical requirement is to re-satisfy the constraints repeatedly as the user moves some part of a figure—each time the screen is refreshed the constraints must be re-satisfied. Each of these constraint satisfaction problems differs from the previous ones only in the values of some of the constants in the constraints (for example, the mouse position). One strategy for achieving the required interactive response times is to compile a constraint satisfaction plan: a block of code that can be executed repeatedly to re-solve the constraints with different input parameters. (We can view this as a kind of partial evaluation of the constraint solving algorithm.) This has long been done for local propagation solvers (e.g. [1]), and more recently for simultaneous linear equations [4] and for acyclic sets of inequality constraints [2]. However, there have not been any systems that can compile plans for systems of constraints including a cyclic set of simultaneous equalities and inequalities. That lack is addressed here.

The original motivation for this work was constraint solving for user interface toolkits and other kinds of graphical interactive systems. The technique is useful for two reasons: first, it produces code that requires no runtime support for constraint solving; second, the code is very efficient. The technique should also be useful in other applications where either of these considerations is a factor.

The ability to get rid of runtime support is important in many real software tasks. One example of this sort is a constraint-based authoring environment for producing Java applets, where the behaviour of the applet is partially specified using constraints. After the applet is written and tested in the authoring environment, the constraint compiler is used to produce straight Java code that can be shipped over the net and run on a remote machine, without requiring a runtime constraint solver on the remote machine. Some preliminary work on such an application has been done [6], and we have used the constraint compiler to produce Java code for several applets. These include an interactive demonstration of a geometric theorem, and an abacus simulation, which were then included in web documents. As a second example, when building an embedded real-time engine controller, predictable performance is needed: compiled code can provide that, but calls to runtime constraint solvers generally cannot. Finally, in developing a product, a company might use constraints in developing the application and not want to ship a proprietary constraint solving package with that application [8].

The compiled code is also very efficient—our measurements show speeds from 5 to 20 times faster than the same test cases performed using a runtime solver based on the simplex algorithm [7]. The runtime solver is reasonably efficient as well, and in many applications the constraint solving time is dominated by the graphics refresh time for either technique. However, in some cases the simplex solver has markedly varying response times—much of the time it is extremely fast, but when a succession of pivots are required it slows down considerably, giving rise overall to a more jerky quality to the interaction. (People prefer uniform response times to varying ones.) The additional speed could be important in other cases as well, for example when the compiled code is to be run on a slower processor.

In this paper we show how we can use Fourier elimination to compile heirarchical constraint solving.[3] In Section 2, we introduce the notion of solving constraints by projection, and show how it can be used to produce compiled code. We review the notion of *constraint hierarchies* for expressing preferential constraints in Section 3, before showing how to adapt the projection algorithm to handle hierarchical constraints in Section 4. In Section 5, we address the issue of redundancy management. In Section 6, we discuss some extensions to the algorithm. We examine some of the compile-time and run-time properties of our prototype implementation in Section 7, while in Section 8 we present some theoretical complexity results for the algorithm. Finally, in Section 9 we conclude.

2. Solving constraints using projection

In this section we briefly illustrate how projection can be used to find solutions to linear equality and inequality constraints. For the moment we will ignore preferential constraints, and also ignore issues of compilation.

A *primitive constraint* in this context is a linear inequality or equality. A *constraint* is a set of primitive constraints. We assume every primitive constraint is written in a simplified form so that no variable appears twice in the same primitive constraint. Let $vars(C)$ denote the set of variables appearing in constraint C, and similarly let $vars(c)$ denote the set of variables appearing in equation or inequality c. We denote by $\bar{\exists}_W F$ the existential quantification of all variables in F except W, that is the formula $\exists v_1 \exists v_2 \cdots \exists v_k\ F$, where W is a set of variables and $\{v_1, \ldots, v_k\} = vars(F) - W$.

For each variable $x \in vars(C)$ we can partition C in the following way. Let C_x^0 be the set of primitive constraints $c \in C$ where $x \notin vars(c)$. Let $C_x^=$ be the set of equations $c \in C$ where $x \in vars(c)$. Let C_x^+ be the set of inequalities $c \in C$ such that c is equivalent to an inequality of the form $x \leq e$, where e is a linear expression not involving x. Finally, let C_x^- be the set of inequalities $c \in C$ such that c is equivalent to an inequality of the form $e \leq x$, again where e is a linear expression not involving x.

The projection algorithm in Figure 1 eliminates a variable x from constraint C and returns constraint $D \leftrightarrow \exists x C$ using either Gaussian elimination or Fourier elimination.

To solve constraints we project out each variable in turn using project until no variables remain. The resulting constraint C' is equivalent to $\bar{\exists}_\emptyset C$. None of the primitive constraints in C' involve any variables and hence we can straightforwardly determine whether C' is satisfiable or not. This answers the satisfiability question

```
project(C, x)
    if exists c ∈ C_x^= where c ≡ x = e then
        D := C − {c} with every occurrence of x replaced by e
    else
        D := C_x^0;
        for each c ∈ C_x^+ where c ≡ x ≤ e^+
            for each c ∈ C_x^- where c ≡ e^- ≤ x
                D := D ∪ {e^- ≤ e^+};
            end for each
        end for each
    end if
    return D;
```

Figure 1. Combined Fourier-Gaussian projection

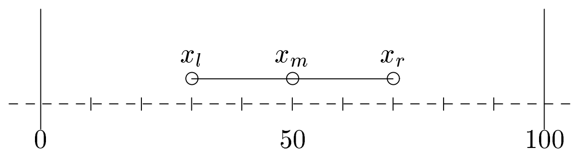


Figure 2. A simple constrained picture

for C. We can then use the intermediate constraints C_i produced after $i-1$ applications of project, as well as the variables $x_1, \ldots, x_n$ in their order of elimination, to build a solution to C. For $1 \leq i \leq n$ it is the case that $vars(C_i) = \{x_i, \ldots, x_n\}$. So C_n only contains variable x_n and hence only contains constraints of the form $x_n = b$, $x_n \leq b$ and $b \leq x_n$. Since C is satisfiable we can find a valuation for x_n which is a solution of C_n.

We can extend a solution $\theta = \{x_{i+1} \mapsto d_{i+1}, \ldots, x_n \mapsto d_n\}$ of C_{i+1} to a solution of C_i as follows. Consider each of the constraints in C_i that involve x_i. If one is an equation equivalent to $x_i = e$, where e is an expression only involving variables $\{x_{i+1}, \ldots, x_n\}$, then we can let $d_i = e\theta$. Otherwise C_i only contains inequalities containing x_i. These can be written in the form $x_i \leq e_j$ or $e_k \leq x_i$, where e_j or e_k is an expression only involving variables $\{x_{i+1}, \ldots, x_n\}$. Using valuation θ we can determine the minimum u of the $e_j\theta$ and maximum l of the $e_k\theta$. Any value for d_i between l and u gives a valid solution of C_i. The new solution is thus $\{x_i \mapsto d_i, x_{i+1} \mapsto d_{i+1}, \ldots, x_n \mapsto d_n\}$.

C_1	C_2	C_3	C_4
$2x_m = x_l + x_r$	$x_m + 5 \leq x_r$	$5 \leq x_m$	$5 \leq 95$
$x_l + 10 \leq x_r$	$2x_m - 100 \leq x_r$	$x_m \leq 95$	
$x_l, x_m, x_r \leq 100$	$x_r \leq 2x_m$		
$0 \leq x_l, x_m, x_r$	$x_m, x_r \leq 100$		
	$0 \leq x_m, x_r$		

Figure 3. Constraint projection example

We now give an example of constraint solving using projection, using the illustration in Figure 2. The constraints are as follows: x_m is constrained to be the midpoint of the line from x_l to x_r, x_l is constrained to be at least 10 to the left of x_r, and all variables must lie in the range 0 to 100.

We can represent this using the constraint C_1 shown in Figure 3. To solve this constraint using variable elimination, we start with C_1, select a variable and project it out, and continue until no variables remain. Suppose we first select x_l. We can project it out using the equation $2x_m = x_l + x_r$, yielding the constraint C_2, and then project out x_r, yielding the constraint C_3 (where we have eliminated some simple redundancy). Finally, eliminating x_m we obtain C_4, which is clearly satisfiable.

We now show how to construct a solution from these constraints. By inspecting C_3, we know we can pick any value between 5 and 95 for x_m, say 50. Next we examine the constraints in C_2 involving x_r. These are $\{x_m + 5 \leq x_r, 2x_m - 100 \leq x_r, 0 \leq x_r, x_r \leq 2x_m, x_r \leq 100\}$. Given $x_m = 50$, this becomes $\{55 \leq x_r, 0 \leq x_r, 0 \leq x_r, x_r \leq 100, x_r \leq 100\}$. We can thus pick any value for x_r in the range [55..100], say 70. Finally, examining the constraints in C_1, there is an equation involving x_l, namely $x_l = 2x_m - x_r$, so we can use this equation directly to set x_l to 30.

We now use this information to compile a sequence of statements that constructs a solution to the constraints, and returns it as a triple, illustrated in Figure 4. As it stands this sequence of statements isn't very interesting, since it only solves one problem. However, in Section 4 we show how to use a similar technique to compile code parameterized by appropriate inputs.

3. Constraint hierarchies

As discussed earlier, for interactive graphics applications, it is important to be able to express preferences as well as requirements in the constraint system; in particular a desire for minimizing change as a figure is manipulated. We use *constraint hierarchies* [5] for specifying the desired solutions to a collection of required and preferential constraints independent of the particular algorithm involved.

```
$x_m^l$ := 5;
$x_m^u$ := 95;
choose $x_m \in [x_m^l..x_m^u]$
$x_r^l$ := max { $x_m + 5$, $2x_m - 100$, 0 };
$x_r^u$ := min { $2x_m$, 100 };
choose $x_r \in [x_r^l..x_r^u]$
$x_l$ := $2x_m - x_r$;
return $(x_m, x_r, x_l)$;
```

Figure 4. Code to solve the constraints of Figure 2, obtained by Fourier elimination

A *labelled primitive constraint* is a primitive constraint labelled with a strength, written sc, where s is a strength and c is a primitive constraint. Strengths are non-negative integers. For clarity, we give symbolic names to the different strengths of primitive constraints. Strength 0, with the symbolic name required, is always reserved for required constraints. For the purposes of this paper we shall use the following symbolic names for strengths: strong $= 1$, medium $= 2$, and weak $= 3$, but in general the approach works for any number of strength levels. Note that a higher integer "strength" indicates a weaker constraint.

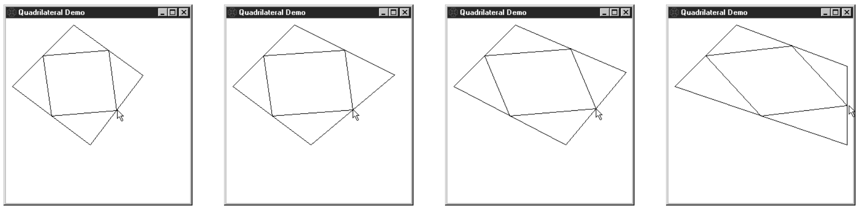
A *constraint hierarchy* is a multiset of labelled primitive constraints. Given a constraint hierarchy H, H_0 denotes the multiset of required primitive constraints in H, with their labels removed. In the same way, we define the multisets $H_1, H_2, \ldots, H_n$ for strengths $1, 2, \ldots, n$.

Since we are now dealing with non-required constraints, valuations of interest may not satisfy all constraints. Hence we need to be able to measure the degree to which a constraint is satisfied. We assume that for each primitive constraint c we have an error function $e(c\theta)$ that returns a non-negative real number indicating how nearly the primitive constraint c is satisfied for a valuation θ. In the work described here, the domain is the reals, and the error function returns a value that varies depending on how nearly the constraint is satisfied. For example, the error for $x + y = z$ is $|x + y - z|$, while the error for $x \leq y$ is $\max(0, x - y)$.

The set S of *solutions* to the hierarchy is defined as follows. S_0 is the set of valuations such that all the H_0 constraints hold; from this we form S by eliminating all potential valuations that are worse than some other potential valuation using the comparator predicate *better*.

$$\begin{aligned} S_0 &= \{\theta \mid \forall c \in H_0 \; c\theta \text{ holds}\} \\ S &= \{\theta \mid \theta \in S_0 \wedge \forall \sigma \in S_0 \; \neg \mathit{better}(\sigma, \theta, H)\} \end{aligned}$$

For now we assume the use of *local comparators* for better, which compare two solutions primitive constraint by primitive constraint. (In Section 6, we will look at how we can extend our approach to certain kinds of global comparators.) A valuation θ is *locally-better* than another valuation σ if, for each of the primitive



(a) (b) (c) (d)

Figure 5. Demonstrating a theorem about quadrilaterals

constraints through to some strength level $k-1$, the error after applying θ is equal to that after applying σ, and at level k the error is strictly less for at least one primitive constraint and less than or equal for all the rest.

$$\begin{aligned}
&\textit{locally-better}(\theta, \sigma, H) \equiv \exists k > 0 \ \text{ such that} \\
&\quad \forall i \in 1 \ldots k-1 \ \forall p \in H_i \ \ e(p\theta) = e(p\sigma) \\
&\quad \wedge \ \exists q \in H_k \ \ e(q\theta) < e(q\sigma) \\
&\quad \wedge \ \forall r \in H_k \ \ e(r\theta) \leq e(r\sigma)
\end{aligned}$$

A locally-better comparator with such an error function is known as *locally-error-better*.

A traditional demonstration of constraint-based graphics is the Quadrilateral Theorem illustrated in Figure 5. The screen snapshots are taken from our Smalltalk implementation, which uses the code produced by the projection algorithm. Each side of the quadrilateral is bisected, and lines are drawn between the midpoints (these inner lines always form a parallelogram). This is expressed as a constraint on each midpoint that it lie halfway between the endpoints of its line.

In addition, all points are constrained to be at least 10 pixels from the sides of the window, the north vertex is constrained to be at least 30 pixels above the south vertex, and the east vertex is constrained to be at least 30 pixels to the right of the west vertex (the latter two so that the quadrilateral cannot collapse to a point). Taken together, these constraints are too difficult for most UI constraint solvers, since they involve cyclic equality and inequality constraints.

In Figure 5(a) we have picked up one of the midpoints with the mouse and begun to move it by temporarily adding an *edit constraint* equating the position of the midpoint and the mouse. This constraint is strongly preferred but not required—we will violate it if necessary. The corner points are weakly constrained to stay where they are by *stay constraints*. The constraint heirarchy representing the quadilateral problem is shown in Figure 6, where (n_x, n_y), (e_x, e_y), (s_x, s_y) and (w_x, w_y) are the coordinates for the north, east, south and west vertices, respectively. The coordinates of the midpoints are (ne_x, ne_y), (se_x, se_y), (sw_x, sw_y) and (nw_x, nw_y). $(\text{Mouse}_x, \text{Mouse}_y)$ are the current coordinates of the mouse, and $\text{old}(n_x)$, $\text{old}(n_y)$,

required	$2ne_x = n_x + e_x$	strong	$se_x = \mathrm{Mouse}_x$
required	$2ne_y = n_y + e_y$	strong	$se_y = \mathrm{Mouse}_y$
required	$2se_x = s_x + e_x$	weak	$n_x = \mathrm{old}(n_x)$
required	$2se_y = s_y + e_y$	weak	$n_y = \mathrm{old}(n_y)$
required	$2sw_x = s_x + w_x$	weak	$e_x = \mathrm{old}(e_x)$
required	$2sw_y = s_y + w_y$	weak	$e_y = \mathrm{old}(e_y)$
required	$2nw_x = n_x + w_x$	weak	$s_x = \mathrm{old}(s_x)$
required	$2nw_y = n_y + w_y$	weak	$s_y = \mathrm{old}(s_y)$
required	$n_y \geq s_y + 30$	weak	$w_x = \mathrm{old}(w_x)$
required	$e_x \geq w_x + 30$	weak	$w_y = \mathrm{old}(w_y)$
required	$10 \leq n_x, e_x, s_x, w_x \leq 290$		
required	$10 \leq n_y, e_y, s_y, w_y \leq 290$		

Figure 6. Constraint heirarchy representing the quadrilateral problem

etc. give the old coordinates of the points (where we want them to stay). The strong edit constraints ensure that the south east midpoint tries to follow the mouse, while the weak stay constraints try to maintain the corner points in their old positions.

In Figure 5(b) the mouse has been moved to the right, and to keep the midpoint constraint satisfied the east vertex (e_x, e_y) has moved as well. We continue moving to the right. In Figure 5(c) the east vertex has run into the imaginary wall resulting from the constraint that it be at least 10 pixels from the window boundary, and can move no further. As a result, in order to maintain the midpoint constraint, the south vertex (s_x, s_y) has begun moving instead. Finally, in Figure 5(d) the mouse has moved beyond the permitted region for the midpoint. The midpoint has moved as close to the mouse as possible, thus causing the two endpoints of its line to be pressed against the boundary as well.

Formally, the stay and edit constraints from this example are simply constraints of the form $v = b$ for variable v and constant b. However, when we come to compile code for repeatedly solving a collection of constraints, it will be important to handle these constraints efficiently, since the value of b will change for each solution: for the stay constraints, it will be the value of v on the previous step; for the edit constraints, it will come from some external input. The other constraints will remain unaltered for each step.

4. Compiling projection for hierarchies

We can now describe the complete algorithm for compiling code that can repeatedly find a locally-error-better solution to a constraint hierarchy given a series of input values. We assume the constraint hierarchy is in *hierarchical normal form.* In this form, the only kind of preferential constraints are ones of the form $v = b$ for a variable v and constant b; all other constraints are required (i.e. we must satisfy them in any solution). In practice the original problem will usually already be in this form. If not, it is easy to transform an arbitrary linear constraint hierarchy into this form by adding error variables. For example, $\mathsf{strong}(e \leq b)$, where e is a linear expression, becomes $\mathsf{required}(e \leq b + v_e) \wedge \mathsf{strong}(v_e = 0)$, where v_e is a new error variable.

Recall that a hierarchical constraint H can be separated into parts H_i, each of which contain the primitive constraints of strength i. If H is in hierarchical normal form, then for $i \geq 1$ H_i will contain only equations of the form $v = b$ where v is a variable and b is a constant. We can use this information to build a strength partitioning of the variables. Let V_i be defined as follows:

$$\begin{aligned} V_1 &= vars(H_1) \\ V_{i+1} &= vars(H_{i+1}) - (V_1 \cup \cdots \cup V_i) \end{aligned}$$

Thus V_i contains those variables whose strongest non-required primitive constraint is of strength i. For simplicity, we assume that every variable is involved in a non-required constraint. This will normally be the case in constraint-based graphics applications. (If not, we can add a very weak stay constraint—weaker than any of the existing constraints—to any variable not otherwise involved in a non-required constraint. Any locally-error-better solution to the new hierarchical constraint will also be a locally-error-better solution to the original constraint.)

Consider solving the constraints for Figure 2 augmented with the preferential constraints $\mathsf{strong}(x_m = b_m) \wedge \mathsf{weak}(x_r = b_r) \wedge \mathsf{weak}(x_l = b_l)$. Then $H_1 = x_m = b_m$ and $H_3 = \{x_r = b_r, x_l = b_l\}$. The strength partitioning of the variables gives $V_1 = \{x_m\}$, $V_2 = \emptyset$ and $V_3 = \{x_r, x_l\}$.

Now we can use the projection algorithm to build code for solving hierarchical normal form constraints, where each problem differs from the others only in the values of the constants b in the non-required primitive constraints. Each problem corresponds to a solution of the constraint heirarchy required to determine the positions of objects for a screen refresh during the manipulation of the diagram.

We apply the projection algorithm to eliminate all variables in the set V_j before eliminating any variables in V_i where $i < j$. A total ordering $x_1 \prec x_2 \prec \cdots \prec x_n$ on the variables $x_1, \ldots, x_n$ in a hierarchical normal form constraint H *respects the hierarchy* if

$$\forall i < j \quad v \in V_i \wedge w \in V_j \;\rightarrow\; w \prec v.$$

Returning to our example of solving the constraints on Figure 2 augmented with $\mathsf{strong}(x_m = b_m) \wedge \mathsf{weak}(x_r = b_r) \wedge \mathsf{weak}(x_l = b_l)$, an ordering that respects the heirarchy is $x_m \prec x_r \prec x_l$.

The full algorithm for generating code takes an ordering of variables $x_1 \prec x_2 \prec \cdots \prec x_n$, and the $\mathsf{required}$ constraint C. To solve the constraint C in conjunction with H_1, H_2, ..., H_k, for each variable $x_i \in V_l$ we select a constraint $x_i = b_i$ from H_l; that is, we arbitrarily choose one of the highest strength non-required constraints on x_i to determine the value b_i. Making such a choice is correct because of the locally-error-better comparator: minimising the error of one of the strongest non-required constraints will always give a locally-error-better so-

```
code_generate(⟨x_1, ..., x_n⟩ , C)
    C_1 := C;
    for i := 1 to n
        C_{i+1} := project(C_i, x_i);
    end for
    for i := n to 1
        if exists c ∈ C_i where c ≡ x_i = e then
            emit( x_i := e );
        else
            minset := ∅; maxset := ∅;
            for each primitive constraint c ∈ C_i where x_i ∈ vars(c)
                if c ≡ e ≤ x_i then
                    minset := minset ∪ {e};
                else if c ≡ x_i ≤ e then
                    maxset := maxset ∪ {e};
                end if
            end for each
            emit( x_i^l := max { minset } );
            emit( x_i^u := min { maxset } );
            emit( if b_i ∈ [x_i^l..x_i^u] then );
            emit( x_i := b_i );
            emit( else if b_i ≤ x_i^l then );
            emit( x_i := x_i^l );
            emit( else x_i := x_i^u );
            emit( end if );
        end if
    end for
    emit( return (x_1, ..., x_n) );
```

Figure 7. Code generation algorithm

lution. (By choosing a different constraint of the same strength, we would compute a different but still valid locally-error-better solution.)

The code produced by the algorithm in Figure 7 will either set x_i to b_i if this is a legitimate choice given the solution determined so far, or set x_i to its lower or upper bound, whichever is closest to the value b_i.

THEOREM 1 (CORRECTNESS OF CODE_GENERATE) *Let C be a constraint with variables $\{x_1, \ldots, x_n\}$. Given a variable ordering $x_1 \prec x_2 \prec \cdots \prec x_n$ for variables in constraint C, the solution $(d_1, \ldots, d_n)$ returned by the code produced by code_generate$(\{x_1, \ldots, x_n\}, C, V)$ will be a locally-error-better solution for the constraint*

$$C \wedge s_1(x_1 = b_1) \wedge \cdots \wedge s_n(x_n = b_n)$$

where for $2 \leq i \leq n$, strength $s_{i-1} \geq s_i > 0$ (i.e. strength s_{i-1} is the same as or weaker than strength s_i).

Proof: The value for each variable is chosen from the available range of values for that variable as determined by Fourier-Gaussian elimination. Thus, by the correctness of Fourier-Gaussian elimination, the algorithm always produces solutions that satisfy the required constraints in C. It remains to show that the solution produced is also locally-error-better.

Assume to the contrary that $(d_1, \ldots, d_n)$ is not a locally-error-better solution to the constraints. Then there exists another solution $(d'_1, \ldots, d'_n)$ that is locally-error-better than $(d_1, \ldots, d_n)$. Let j be the largest index where $d'_j \neq d_j$. We examine the selection of the value of the variable x_j. By the correctness of Fourier-Gaussian elimination, $\bar{\exists}_{\{x_j, \ldots, x_n\}} C \;\leftrightarrow\; C_j$ and hence

$$(\bar{\exists}_{\{x_j, \ldots x_n\}} C) \wedge x_{j+1} = d_{j+1} \wedge \cdots \wedge x_n = d_n \;\leftrightarrow\; C_j \wedge x_{j+1} = d_{j+1} \wedge \cdots \wedge x_n = d_n$$

Given that the variables $x_{j+1}, \ldots, x_n$ take the values $d_{j+1}, \ldots, d_n$ respectively, then clearly the only solutions of x_j in C_j are in the range $[x^l_j..x^u_j]$.

If $b_j \in [x^l_j..x^u_j]$ then $d_j = b_j$ and $d'_j \neq b_j$. Hence the error for the equation $x_j = b_j$ is greater for the solution $(d'_1, \ldots, d'_n)$ than for $(d_1, \ldots, d_n)$. But this means $(d'_1, \ldots, d'_n)$ is not locally-error-better than $(d_1, \ldots, d_n)$ since it has a greater error in the equation $x_j = b_j$ and $(d_1, \ldots, d_n)$ has the same error for every non-required equation of higher strength.

Otherwise, if $b_j < x^l_j$ then $d_j = x^l_j$ and so $d'_j > d_j$ (since d'_j must be in the range $[x^l_j..x^u_j]$). Again the error for the constraint $x_j = b_j$ is greater for $(d'_1, \ldots, d'_n)$ than for $(d_1, \ldots, d_n)$, which implies that $(d'_1, \ldots, d'_n)$ is not locally-error-better than $(d_1, \ldots, d_n)$. The remaining case of $b_j > x^u_j$ is similar. ■

Continuing with our running example of compiling the constraints for Figure 2 augmented with $\mathsf{strong}(x_m = b_m) \wedge \mathsf{weak}(x_r = b_r) \wedge \mathsf{weak}(x_l = b_l)$, the resulting code is shown in Figure 8. Note the very high similarity of this code with that of Figure 4.

Suppose the midpoint of the line is selected by the mouse to move to position 60, and the remaining points are constrained to stay where they are ($x_l = 30$ and $x_r = 70$). This then imposes constraints $\mathsf{strong}(x_m = 60) \wedge \mathsf{weak}(x_r = 70) \wedge \mathsf{weak}(x_l = 30)$, for which the code in Figure 8 generates the answer $(60, 70, 50)$. If the mouse now moves to position 70, the edit and stay constraints translate as $\mathsf{strong}(x_m = 70) \wedge \mathsf{weak}(x_r = 70) \wedge \mathsf{weak}(x_l = 60)$ and the code generates the answer $(70, 75, 65)$. If the mouse now moves to position 0, the code generates the answer $(5, 10, 0)$.

5. Redundancy management

One of the problems with Fourier elimination is that a large number of constraints can be produced (potentially an exponential number), many of which are redundant. For a constraint compiler based on Fourier elimination to succeed, it is crucial

```
x_m^l := 5;
x_m^u := 95;
if b_m ∈ [x_m^l..x_m^u]
     x_m := b_m;
else if b_m < x_m^l
     x_m := x_m^l;
else x_m := x_m^u;
x_r^l := max { x_m + 5, 2x_m − 100, 0 };
x_r^u := min { 2x_m, 100 };
if b_r ∈ [x_r^l..x_r^u]
     x_r := b_r;
else if b_r < x_r^l
     x_r := x_r^l;
else x_r := x_r^u;
x_l := 2x_m − x_r;
return (x_m, x_r, x_l);
```

Figure 8. Code for the midpoint example constraints

that the issue of redundant constraints be addressed. Detecting and eliminating all redundancy is not very practical, but there are a number of types of redundancy that are cheap to detect, while still being effective at keeping the number of redundant constraints down.

One of these is *quasi-syntactic redundancy.* A primitive constraint

$$c_1 \equiv a_1x_1 + \ldots + a_mx_m \leq a_0$$

is quasi-syntactically redundant with respect to

$$\begin{aligned} & c_2 \equiv a_1x_1 + \ldots + a_mx_m \leq b_0 \\ \text{or} \quad & c_2 \equiv a_1x_1 + \ldots + a_mx_m = b_0 \end{aligned}$$

if $b_0 \leq a_0$; if this is the case c_1 can be dropped without affecting the result. This test is inexpensive, $O(n \log n)$ for testing n primitive constraints, yet allows us to get rid of a significant number of redundant constraints.

The other main class of redundancy which we detect and eliminate targets the output constraints. During the projection process, the best known (constant) bounds are maintained for each variable. Then when a variable (say x) is to be eliminated, the constraints that are to yield upper and lower bounds for x are examined. Any constraint that can be shown to be redundant with respect to another, by using the bounds information accumulated earlier, is discarded. Doing this redundancy elimination at this point can drastically reduce the size of the cross-product formed during the elimination of x, and also directly reduces the number of compiled constraints.

The bounds information is used by evaluating the minimum and maximum values an expression can take, subject to the known bounds. Best use is made of this by examining the expressions for, say, the upper bounds on x in a pairwise fashion, and evaluating the difference between the expressions. If the minimum possible value of the difference is non-negative, the first expression is guaranteed to be no smaller than the second; if the maximum value is non-positive, it is guaranteed to be no larger. In both of these cases, one of the constraints may be discarded as redundant; otherwise, no redundancy information can be deduced. By evaluating the difference of the constraints in this fashion, rather than each constraint individually, one is able to obtain much more redundancy information. For example, suppose we have bounds on x of $x \leq 2y$, $x \leq y + 10$, and we know that $-5 \leq y \leq 5$. For the first bound, we can deduce that it lies between -10 and 10; for the second it is between 5 and 15. Since $5 < 10$, we cannot guarantee that the first constraint will always be tighter than the second. However, if we look at the difference $(y - 10)$, we can work out that it lies between -15 and -5, and thus *can* guarantee that the first constraint will always be tighter than the second, and so can discard the latter as redundant.

The second main approach to managing redundancy is to try to minimise the number of redundant constraints generated in the first place. This can be done by being clever about the order in which variables are eliminated. The algorithm as it stands leaves us reasonable freedom in the choice of which variable to eliminate (we can choose variables at the same strength in any order we like). Some simple heuristics about which variable to eliminate next can make a substantial difference to the number of constraints generated. An obvious and useful heuristic is to choose a variable appearing in an equation, so that Gaussian elimination can be used rather than Fourier. If there are no such variables, choosing x that minimises $|C_x^+| \times |C_x^-|$ can work well (this minimises for the next step the sum of the number of output constraints and the number of constraints remaining, prior to redundancy elimination). The actual heuristic we use counts the number of linear terms in the constraints, rather than the number of constraints, since this discourages the formation of constraints with many terms, which are particularly problematic for Fourier elimination.

We have based a more reasoned approach to selecting which variable to eliminate next on the results of Theorem 4 (see Section 8). This often allows us to process relatively small groups of variables and constraints at any given time, which limits the scope for producing redundant constraints. In practice, this approach seems to work well.

6. Refinements

In this section we discuss some refinements to the solver that would appear to increase the class of problems it can solve, but can be implemented simply by altering the input to the main algorithm.

$$
\begin{array}{rl}
\textsf{required} & 2x_m = x_l + x_r \\
\textsf{required} & x_l + 10 \leq x_r \\
\textsf{required} & x_l \geq 0 \\
\textsf{required} & x_m \geq 0 \\
\textsf{required} & x_r \geq 0 \\
\textsf{required} & x_l \leq 100 \\
\textsf{required} & x_m \leq 100 \\
\textsf{required} & x_r \leq 100 \\
\textsf{strong} & x_l = X \\
\textsf{weak} & x_l = L \\
\textsf{weak} & x_m = M \\
\textsf{weak} & x_r = R
\end{array}
$$

Figure 9. Midpoint example—local comparator version

6.1. Optimising a function

Rather than just returning any solution satisfying the input constraints, it is possible to optimise a linear objective function. This can be done by introducing a new variable representing the value of the objective function, plus a required constraint relating it to the variables it depends on. Then a constraint of an appropriate strength can try to set the value of the objective function to some suitably large (maximise) or small (minimise) value.[4]

Note that one can actually have as many objective functions at as many different strengths as one wants. For instance, one could optimise a particular function subject to only the required constraints, then optimise another subject to the first remaining optimal, then add some non-required constraints on the values of some variables, and then apply a final optimisation: all by assigning appropriate strengths to each. Note that normally an objective function would not share its strength with anything else. For example, if two objective functions were assigned the same strength, the solver would be free in its choice of which to optimise first. Allowing such a choice would not usually make sense unless the objective functions were independent of each other.

6.2. Global comparators

In Sections 3 and 4, we assumed that we were working with local comparators. However, it is possible to achieve an effect equivalent to using any linear global comparator. The way this is done is similar to the introduction of an objective function above, except that this time the function is in terms of error variables. This may require the introduction of extra error variables which were not needed before, as well as some required non-negativity constraints on the error variables.

required	$2x_m = x_l + x_r$	required	$e_l^+ \geq 0$
required	$x_l + 10 \leq x_r$	required	$e_l^- \geq 0$
required	$x_l \geq 0$	required	$e_m^+ \geq 0$
required	$x_m \geq 0$	required	$e_m^- \geq 0$
required	$x_r \geq 0$	required	$e_r^+ \geq 0$
required	$x_l \leq 100$	required	$e_r^- \geq 0$
required	$x_m \leq 100$	required	$e_l^+ + e_l^- + e_m^+ + e_m^- + e_r^+ + e_r^- = g$
required	$x_r \leq 100$	medium	$g = 0$
strong	$x_l = X$	weak	$x_l = L$
required	$x_l + e_l^- = L + e_l^+$	weak	$x_m = M$
required	$x_m + e_m^- = M + e_m^+$	weak	$x_r = R$
required	$x_r + e_r^- = R + e_r^+$		

Figure 10. Midpoint example—global comparator version

Consider the simple midpoint example again, where we have added weak stay constraints on x_l, x_m and x_r, plus a strong constraint on x_l. These constraints are shown in Figure 9. Suppose we wish to minimise the sums of the errors in the weak constraints, with medium strength. The error in the weak constraint $x_r = R$ is $|x_r - R|$. We cannot encode this error directly using a linear constraint due to the modulus. Instead, we introduce a pair of new variables to represent the error: one for when the difference is negative, and one for when it is positive. If each of these variables is then constrained to be non-negative, their sum can be used to represent the error.[5] Thus we have $x_r + e_r^- = R + e_r^+$, with $e_r^+ \geq 0$ and $e_r^- \geq 0$. Taking into consideration all the weak constraints, the global error is then $e_l^+ + e_l^- + e_m^+ + e_m^- + e_r^+ + e_r^- = g$. To minimise this, we also add, with medium strength, $g = 0$. The final result is shown in Figure 10. Note that we have kept the weak stay constraints, to ensure that the variables are set to particular values.

As can be seen, and not too surprisingly, using global comparators comes at a cost, in terms of the size and number of the constraints to be compiled.

7. Computational results

The projection-based compilation algorithm has been implemented and tested. Our prototype implementation is in Mercury [14], and includes a module that is easily adapted to generate code for different target languages. The current implementation produces Smalltalk code, which is stored in a file. Then, in the Smalltalk environment, the code is loaded and incorporated into a graphics application for execution. The advantages of using Smalltalk are that Smalltalk includes an extensive graphics library, making it easy to test interactive graphical programs, and also that we have a Smalltalk implementation of Cassowary, a simplex-based solver, which allows a head-to-head comparison of the run times of the two algorithms.

We have investigated the performance of our algorithm for several medium-sized examples of constraints for interactive graphics.

The *boxcars* benchmark has a number of boxes in a horizontal row. Each box is constrained to be to the right of the previous one, and all are constrained to stay within a specified horizontal range.

The *binary tree* benchmark is a complete binary tree of a given height. Each pair of children are constrained to be at the same height, both must be at least some minimum distance below their parent, and they must be separated from each other by some minimum distance. All parent nodes must be centred over their children, and finally every node must lie within a certain bounding box. This formulation has some redundancy—we could have specified only that the left child be the minimum distance below the parent, rather than both the left and right children, since the children are at the same height. However, this redundancy arises naturally in the specification.

The *grid* benchmark is an $n \times n$ grid of points where every point is constrained to be on an imaginary vertical line through the points above and below it, and on a horizontal line through the points to the left and right. No point can be within a given distance of its neighbours, and all points lie within a given box. Again this specification leads to redundancy in the constraints.

The compilation results are shown in Table 1. For each benchmark we give the following information:

- the number of variables in the original formulation;
- the number of primitive constraints in the original formulation;
- the typical number of primitive constraints in the compiled code (that is, the number of inequalities used for *minset* and *maxset* calculations, plus the equations used to emit calculation code, for a typical choice of edit variable(s)); and
- the typical CPU time required for the Mercury program to produce the Smalltalk code

The compilations were done on a DEC AlphaServer 8400 with eight 300 MHz 21164 processors.

The compiled versions did not suffer from an exponential blow-up in the number of primitive constraints, and indeed all contain fewer constraints than were in the input, the reduction being due to the redundancy elimination performed during projection.

The code produced is extremely efficient, and also has predictable performance irrespective of the input values. Table 2 gives some measurements of the execution speed of the compiled code as compared with the execution speed of the Cassowary constraint solver [7], which uses an efficient simplex-based algorithm specifically adapted for repeatedly solving constraints arising in interactive graphics applications. All timings were done using OTI Smalltalk Version 4.0, running on an IBM Thinkpad 760EL laptop computer.

Table 1. Compilation statistics

Problem	size	variables	primitive constraints	compiled constraints	time (secs)
boxcars	50	50	149	100	0.7
	100	100	299	200	1.4
	200	200	599	400	4.5
	400	400	1199	800	15.8
	800	800	2399	1600	62.0
binary tree	5	62	199	111	0.5
	6	126	407	230	1.2
	7	254	823	470	3.7
	8	510	1655	950	12.5
	9	1022	3319	1910	46.2
grid	5×5	50	180	60	0.4
	10×10	200	760	220	3.3
	20×20	800	3120	840	48.1

Table 2. Runtime statistics

Problem	Time (milliseconds)			Avg. # of pivots
	Fourier	Cassowary	Graphics	(Cassowary only)
50 boxcars (infrequent collisions)	1.0	5.0	38.0	0.11
50 boxcars (frequent collisions)	1.0	16.0	38.0	0.51
depth 5 tree (frequent collisions)	5.3	68.6	46.0	1.55
depth 5 tree (v. frequent collisions)	5.3	124.5	46.0	3.77

The times shown are the average number of milliseconds to perform one update, i.e. to solve the constraint problem with a new value for the mouse position. The Fourier and Cassowary times are both without graphics refresh; the additional refresh time is shown in the "Graphics" column. Finally, for Cassowary, the last column gives the average number of simplex pivots per update. There are two different runs given for the boxcars example, one with the input varying slowly, so that collisions between the boxcars are relatively infrequent, and another with more rapidly varying input, causing more collisions. The time to run the Fourier code is the same for both cases; however, with frequent collisions the Cassowary time per update increases substantially. The reason is that a collision generally requires a pivot in the simplex tableau, which is expensive; when there is no collision, the tableau can be updated very efficiently with no pivoting required. There are also two runs for the tree example. We used a tree with the node displays closely spaced, so that there were many collisions (a moving node bumping into a stationary one) in both cases; the second run involved moving the mouse more quickly to generate even more collisions.

For these examples the Fourier code is approximately 5 to 20 times faster than the runtime simplex solver. For the relatively simple constraints of the boxcars example, the graphics refresh time is substantially more than the time required to satisfy the constraints in any of the tests. However, when using the simplex solver for the tree example, the constraint solving time becomes significant compared to the graphics refresh time. In addition, the simplex solver has variable solving time, which is quite apparent when the mouse is moved very quickly on this example—often the update is extremely fast, but when numerous pivots are needed it slows down, giving a less pleasant jerky quality to the interaction. In fairness to the simplex approach, it should be possible to extrapolate the current direction of the mouse movement and pre-solve some of the pivots; but this has not been done in any of the current systems. In addition, for use with real-time systems the predictable response provided by the compiled Fourier code is essential.

8. Complexity analysis

Most variable elimination algorithms have bad worst case complexities. Projecting one variable from a system of m linear inequalities produces $O(m^2)$ inequalities in the worst case. Hence eliminating n variables from m primitive constraints can be $O(m^{2^n})$. However, our empirical results so far have shown quite reasonable performance for practical problems. In this section we attempt to analyse the situation and point out reasons for this.

One major factor in the reasonable performance of our algorithm is that in many practical problems each primitive constraint only involves a small number of variables, and hence the worst case does not arise. There are also a number of restricted cases that do have much more reasonable worst case complexities. One such restricted case is when each constraint involves at most 2 variables. The *grid* benchmark above falls into this category. The following result is due to Nelson [12].

THEOREM 2 *Let C be a set of m inequalities involving n variables where each inequality involves at most 2 variables. Fourier elimination is $O(mn^{\lceil 2\log n\rceil+3}\log n)$.*

Another example where the worst case phenomena cannot occur is when almost all of the constraints are equations.

THEOREM 3 *Let C be a constraint involving n variables, m linearly independent equations, and l inequalities. Then there is a choice of variable elimination order where Fourier elimination is $O(nm(m+l)+l^{2^{n-m}})$.*

Proof: Use the equations to eliminate m variables, leaving l inequalities in $n-m$ variables. The result follows. ■

It also appears that, in many practical constraint problems in interactive graphics, the constraints are not tightly connected. We examine a class of constraint graph which is not tightly connected. A constraint graph for a constraint C is an undirected graph constructed as follows. There are nodes for each variable in $vars(C)$ and each primitive constraint $c \in C$. There is an edge between variable node v and primitive constraint node c if $v \in vars(c)$. Two nodes x and y in a graph G are *bi-connected* if there exist two node-distinct paths in G from x to y. A bi-connected component of a graph G is a maximal set of nodes N such that each pair $x, y \in N, x \neq y$ is bi-connected in G. A constraint graph G is *k-bi-connected* if there are no bi-connected components of G with more than k variable nodes. The *binary tree* benchmark is an example where the constraint graph is 3-bi-connected.

THEOREM 4 *Let C be a constraint involving n variables and m primitive constraints whose constraint graph is k-bi-connected. Then there is a choice of variable elimination order where a slightly modified Fourier elimination algorithm with quasi-syntactic redundancy elimination is $O(nm^{2^k})$.*

Proof: We start by forming each bi-connected component of the graph into a cluster. These clusters form a tree (or a forest, if the original graph was not fully connected). We proceed to eliminate the clusters one at a time, starting at the leaves.

A leaf cluster will be connected to the rest of the tree via a single node. If that node represents a variable, say x, our task is easy. We simply project all the constraints in the cluster onto x. The only possible resultant constraints on x are bounds, and using quasi syntactic redundancy there can be at most two: $l \leq x$ and $x \leq u$. These bounds are added to the parent cluster.

If, on the other hand, the connecting node represents a constraint, we introduce a temporary variable t and split the constraint first. Let the constraint be of the form $e_1 + e_2$ *op* 0, where *op* is either $\leq$ or $=$, e_1 is a linear expression containing only variables included in the cluster, and e_2 is a linear expression containing no variables from the cluster. Then the constraint is split into $e_1 + t$ *op* 0 and $e_2 - t = 0$, with the former being added to the current leaf cluster, the latter replacing the original constraint, and the variable t being added to the parent cluster. We then project all the constraints in the current leaf cluster onto t, and add the resulting bounds to the parent cluster, as before.

Note that the introduction of temporary variables can increase the number of variable nodes in a cluster. Each temporary variable appears in exactly one equation and at most two bounds constraints, though we note that more than one temporary variable may appear in the same equation (if the original constraint was a cut node where several bi-connected components met). If we eliminate the temporary variables first, it is easy to see that each equation involving temporary variables yields at most two inequalities involving the remaining variables, and that this elimination is $O(l)$, where l temporary variables were involved. This then leaves a cluster with at most k variables to which the standard Fourier elimination will be applied.

Eliminating the variables in a leaf cluster never adds more constraints to the parent cluster than were eliminated in the leaf; hence the maximum number of constraints in any cluster at any given time is m. The maximum number of variables eliminated in any cluster when the standard algorithm is applied is k. Hence processing one cluster with the standard algorithm is $O(m^{2^k})$. There are at most n clusters in the tree, and thus at most n temporary variables introduced. Hence the complexity result holds. ■

Theorem 4 in particular constrains the order in which variables can be eliminated. In interactive graphics problems there are typically two variables with edit constraints (the x and y coordinates of a point being moved). These variables must be eliminated last. This is compatible with the ordering required by the theorems if the x and y constraints are independent (as is the case in all our examples, and in many other cases as well). However, if the problem includes constraints relating the x and y coordinates (e.g. that a shape be a square), then we may be unable to use the complexity guarantees provided by the theorems.

9. Conclusions

Fourier elimination can be used to generate very fast, constraint-free code to solve problems arising in interactive graphical applications. The approach is useful for applications such as real-time systems which need predictable performance, for smoothing the response time in an interactive system, for producing applications that can be run without employing a runtime constraint solver, and when execution speed is very important.

The same approach of compiling constraint solving by projection can be applied to any constraint domain that has a variable elimination function that projects one variable out of a constraint, along with a method for finding solutions of a conjunction of constraints in one variable. Constraint domains that meet these requirements include Boolean constraints, unit two variable per inequality (integer) constraints [11], and partial order constraints.

The current algorithm is a batch one. A direction for future research is the design of an incremental version, which would reuse part of a previous solution when accommodating changes in the constraint (beyond simply changing the constants in the $v = b$ constraints).

Some previous work has involved hybrid constraint solving algorithms, which partition a set of constraints into regions that can then be turned over to an appropriate sub-solver for that class of constraint and constraint topology [3]. Compilation based on Fourier elimination is a promising candidate for use in this architecture, to handle cyclic collections of linear equality and inequality constraints.

Theorem 4 is particularly interesting, because it shows how the structure of the constraints to be solved can be exploited to make the actual solving more efficient. Sabin and Freuder [13] have also studied how the structure of the constraint graph can be exploited; in their case, they demonstrate substantial improvements in the solving time of random binary CSPs, by identifying (an approximation of) the cycle cutset of the constraint graph. This seems a very interesting area to explore further.

Acknowledgements

This project has been funded in part by National Science Foundation Grant IIS-9975990 and by Australian Research Council Large Grant A10017012. Alan Borning's visit to the University of Melbourne was sponsored in part by a Fulbright award.

Notes

1. Current address wh@icparc.ic.ac.uk, IC-Parc, William Penney Laboratory, Imperial College, Exhibition Road, London SW7 2AZ, United Kingdom
2. Note that while user interface objects will be mapped to particular pixel locations on the display, and thus their coordinates should, in theory, be integral, in practice this is usually not necessary (or desirable). For most applications, rounding real coordinates to the nearest integer point is quite appropriate, so locations on the screen are treated as reals to make the constraint solving easier. In this work we assume coordinates are reals. We note that non-linear constraints do arise in user interface applications for such attributes as areas and angles, and finite domain constraint also arise in handling fonts. In such cases more sophisticated techniques are required (e.g. [9]).
3. An earlier version of this paper appears as [10].
4. $\pm$ `MAXFLOAT` or similar should be sufficient for most applications, but if not, it would be trivial to change the solver to implement this feature directly.
5. Strictly, their sum is only an accurate representation of the error when at least one of the variables is zero, but this will be guaranteed by minimising the global error.

References

1. Alan Borning. The programming language aspects of ThingLab, a constraint-oriented simulation laboratory. *ACM Transactions on Programming Languages and Systems*, 3(4):353–387, October 1981.
2. Alan Borning, Richard Anderson, and Bjorn Freeman-Benson. Indigo: A local propagation algorithm for inequality constraints. In *Proceedings of the 1996 ACM Symposium on User Interface Software and Technology*, pages 129–136, Seattle, November 1996.
3. Alan Borning and Bjorn Freeman-Benson. The OTI constraint solver: A constraint library for constructing interactive graphical user interfaces. In *Proceedings of the First International Conference on Principles and Practice of Constraint Programming*, pages 624–628, Cassis, France, September 1995.
4. Alan Borning and Bjorn Freeman-Benson. Ultraviolet: A constraint satisfaction algorithm for interactive graphics. *Constraints: An International Journal*, 3(1):9–32, April 1998.
5. Alan Borning, Bjorn Freeman-Benson, and Molly Wilson. Constraint hierarchies. *Lisp and Symbolic Computation*, 5(3):223–270, September 1992.
6. Alan Borning, Richard Lin, and Kim Marriott. Constraints for the web. In *Proceedings of Fifth ACM International Multi-Media Conference*, pages 173–182, November 1997.
7. Alan Borning, Kim Marriott, Peter Stuckey, and Yi Xiao. Solving linear arithmetic constraints for user interface applications. In *Proceedings of the 1997 ACM Symposium on User Interface Software and Technology*, pages 87–96, October 1997.
8. Bjorn Freeman-Benson. Object Technology International, Personal communication.
9. Daniel H. Greene and F. Frances Yao. Finite-resolution computational geometry. In *Proceedings of the 27th Annual Symposium on the Foundations of Computer Science*, pages 143–152. IEEE Computer Society Press, 1986.
10. W. Harvey, P.J. Stuckey, and A. Borning. Compiling constraint solving using projection. In G. Smolka, editor, *Proceedings of the Third International Conference on Principles and Practices of Constraint Programming*, LNCS, pages 491–505. Springer-Verlag, October 1997.
11. J. Jaffar, M. J. Maher, P.J. Stuckey, and R.H.C. Yap. Beyond finite domains. In *Proceedings of the International Workshop on Principle and Practices of Constraint Programming*, number 874 in LNCS, pages 86–93, Orcas Island, Washington, May 1994. Springer-Verlag.
12. C.G. Nelson. An $n^{\log n}$ algorithm for the two-variable-per-constraint linear programming satisfiability problem. Technical Report STAN-CS-78-689, Stanford University, 1978.
13. Daniel Sabin and Eugene C. Freuder. Understanding and improving the MAC algorithm. In G. Smolka, editor, *Proceedings of the Third International Conference on Principles and Practices of Constraint Programming*, LNCS, pages 167–181. Springer-Verlag, October 1997.

14. Zoltan Somogyi, Fergus Henderson, and Thomas Conway. The execution algorithm of Mercury: an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29(1–3):17–64, October–December 1996.