On Finding Graphically Plausible Solutions to Constraint Hierarchies: The Split Stay Problem

Alan Borning and Greg J. Badros
Dept. of Computer Science and Engineering
University of Washington, Box 352350
Seattle, Washington 98195-2350, USA
+1 206 543 6678
{borning,gjb}@cs.washington.edu

August 2000

Abstract

The semantics of a constraint system for use in an interactive graphical application should accomodate state and change properly. Constraint hierarchies are one technique for expressing a semantics for hard and soft constraints, which is well-suited for this domain. The constraint hierarchy framework is parameterized by a *comparator*, which compares how well two potential solutions to the hierarchy satisfy the soft constraints. Two special varieties of constraints for interactive graphics are *stay constraints* and *edit constraints*. Stay constraints indicate that we prefer that a variable retain its previous value, unless there is a stronger reason for it to be changed. Edit constraints accomodate a sequence of changing inputs from an external device such as a mouse. A number of incremental algorithms have been developed for finding a solution to a constraint hierarchy, which include efficient treatment of stay and edit constraints. After a short review of the literature in this area, this paper discusses an issue regarding constraint hierarchies and interactive graphics that has up to now been solved only in an ad hoc fashion: the *split stay problem*. This problem arises for compound objects such as 2-d points. We prefer solutions that completely satisfy the stay constraints on the x and y components of a given point over solutions that consider the x and y components separately, for example satisfying the constraints on the x part of one point and the y part of another. Unfortunately, some comparators (including ones that turn out to be particularly useful) do not discriminate among solutions in this way. We present a new comparator that does correctly handle the split stay problem.

Keywords: constraint hierarchies, interactive graphics, stay constraints, Cassowary

1 Background

Constraints have been used in interactive graphical applications since Sketchpad in the early 60's [11]. Constraints in this domain, as in others, provide a declarative way for the user to specify what is desired rather than how to achieve it. Two particular features of this domain, however, stem from interactivity. First, we need to handle the semantics of state and change properly when a graphical figure is manipulated. Second, to preserve interactive response we need efficient incremental algorithms for satisfying the constraints.

Constraint hierarchies provide one technique for specifying the desired solutions to a collection of hard and soft constraints, which has proven convenient for this domain. (A short description is given in Section 2.) We can use hard (required) constraints to specify properties of the graphical figure that must be maintained, and soft constraints for the preferences.

In particular, in interactive graphical applications one typically wants parts of the figure to remain fixed, unless there is some reason for them to change. This desire can be expressed using soft *stay* constraints. A stay constraint is one of the form $v = c$ for variable v and constant c, where v is typically the x or y part of a point, and c is the value of x or y at the previous time step. The stay constraints will usually be weak, so that they can be overridden by stronger constraints (such as a constraint that a point follow the mouse when editing).

Another use for soft constraints is to express the intent that a point follow the mouse (or more generally, that some variable is repeatedly set to a new externally imposed value). This so-called *edit constraint* is also of the form $v = c$ for variable v and constant c. Here, however, c comes from some external source (e.g. the x or y position of the mouse). Each time the screen is refreshed while moving a part, we will want to find a new solution to the hard and soft constraints given new values for the constants in these edit constraints. By making the edit constraint strong but not required, we can correctly handle conditions such as the cursor being outside of the permitted area of movement. (Typically, the point being moved will be placed as close to the cursor as possible while remaining within its permitted range.)

A number of incremental algorithms have been developed for solving constraint hierarchies, with special treatment for edit and stay constraints, including DeltaBlue [10], QuickPlan [12], SkyBlue [9], Detail [7], and Cassowary [2, 6]. Cassowary in particular can solve arbitrary sets of linear equality and inequality constraints.

A recent application of Cassowary is constraint-based web layout [3]. Here, we extend the Cascading Style Sheet standard [8] to include constraints. For example, we can constrain two columns in a table to have the same width, and to be at least 10 times as wide as the gutter between them, or constrain a heading font to be 2 points larger than the body font. These constraints can be preferences as well as requirements, and can come from the viewer as well as the page author. (For example, the viewer might insist on a minimum of 12 point font sizes, or a particular browser width.) The final appearance of the web page is thus the result of an arbitration between the desires of the author and viewer, where this arbitration is performed by solving the combined set of constraints.

A second application is the Scheme Constraints Window Manager [4]. This window manager allows users to express their intentions using both direct manipulation and with persistent constraints, e.g. that one window be always to the left of another and of the same height. A number of other features support the ready use of constraints in the window manager, including a graphical toolbar for applying constraints, a recorder for defining new compound constraints using a simple programming-by-demonstration technique, and a graphical constraint investigator for visually display the current set of constraints when desired.

Finally, a third application extends the Scalable Vector Graphics (SVG) standard for the web [1] to allow constraints on the positions of graphical elements. SVG is an important advance over including bitmap images in web pages, since the image is resolution-independent. However, the internal geometry of an SVG image is fixed. Constraint Scalable Vector Graphics generalizes SVG, so that a given illustration can be resized and reformatted appropriately for the available space, while still satisfying its constraints. In addition, constraints on a read-only "time" variable provide a simple animation facility.

2 Constraint Hierarchy Theory

Briefly, we can define a constraint hierarchy as follows. (See Reference [5] for a more complete discussion.)

A *labeled constraint* is a constraint labeled with a strength, written sc, where s is a strength and c is a constraint. Strengths will be integers between 0 and n, where n is the number of non-required levels. For clarity, we often give symbolic names to the different strengths. Strength 0, with the symbolic name required, is always reserved for required (hard) constraints.

A *constraint hierarchy* is a multiset of labeled constraints. Given a constraint hierarchy H, H_0 denotes the required constraints in H, with their labels removed. In the same way, we define the sets $H_1, H_2, \ldots, H_n$ for levels $1, 2, \ldots, n$. We also define $H_k = \emptyset$ for $k > n$.

A *solution* to a constraint hierarchy H is a valuation for the free variables in H, i.e., a function that maps the free variables in H to elements in the domain $\mathcal{D}$. We wish to define the set S of all solutions to H. Clearly, each valuation in S must be such that, after it is applied, all the required constraints hold. In addition, we desire each valuation in S to be such that it satisfies the non-required constraints as well as possible, respecting their relative strengths. To formalize this desire, we first define the set S_0 of valuations such that all the H_0 constraints hold. Then, using S_0, we define the desired set S by eliminating all potential valuations that are worse than some other potential valuation using the comparator predicate *better*. (In the definition, $c\theta$ denotes the boolean result of applying the valuation θ to c, and we say that "$c\theta$ holds" if and only if $c\theta = \mathbf{true}$.)

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

There are many plausible candidates for comparators. While we are primarily concerned with one particular comparator in this paper (*locally-metric-better*), some of the others will arise as well.

We now define several different comparators. Note that in general, *better* will not provide a total ordering—there may exist θ and σ such that θ is not better than σ and σ is not better than θ. In the definitions, we will need an error function $e(c\theta)$ that returns a non-negative real number indicating how nearly constraint c is satisfied for a valuation θ. This function must have the property that $e(c\theta) = 0$ if and only if $c\theta$ holds. For any domain $\mathcal{D}$, we can use the trivial error function that returns 0 if the constraint is satisfied and 1 if it is not. A comparator that uses this error function is a *predicate* comparator. For a domain that is a metric space, we can use its metric in computing the error instead of the trivial error function. Such a comparator is a *metric* comparator. For example, for $\mathcal{R}$ the error for $x = y$ is $|x - y|$, while the error for $x \leq y$ is $x - y$ if $x > y$ and otherwise 0.

The first of the comparators, *locally-better*, considers each constraint in H individually.

Definition. A valuation θ is *locally-better* than another valuation σ if, for each of the constraints through some 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 constraint and less than or equal for all the rest.

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

Next, we define a schema *globally-better* for global comparators. The schema is parameterized by a function g that combines the errors of all the constraints H_i at a given level.

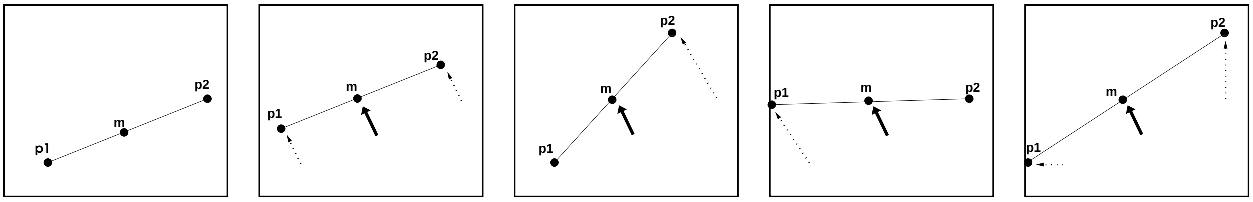
Definition. A valuation θ is *globally-better* than another valuation σ if, for each level through some level $k-1$, the combined errors of the constraints after applying θ is equal to that after applying σ, and at level k it is strictly less.

$$\begin{array}{l} \mathit{globally\text{-}better}(\theta, \sigma, H, g) \equiv \\ \quad \exists k > 0 \;\; \text{such that} \\ \quad\quad \forall i \in 1 \ldots k-1 \;\; g(\theta, H_i) = g(\sigma, H_i) \\ \quad\quad \wedge \;\; g(\theta, H_k) < g(\sigma, H_k) \end{array}$$

Using *globally-better*, we now define two global comparators, using different combining functions g. The weight for constraint p is denoted by w_p. Each weight is a positive real number.

$$\begin{array}{rcl} \mathit{weighted\text{-}sum\text{-}better}(\theta, \sigma, H) & \equiv & \mathit{globally\text{-}better}(\theta, \sigma, H, g) \\ \text{where} \;\; g(\tau, H_i) & \equiv & \displaystyle\sum_{p \in H_i} w_p e(p\tau) \\[2ex] \mathit{least\text{-}squares\text{-}better}(\theta, \sigma, H) & \equiv & \mathit{globally\text{-}better}(\theta, \sigma, H, g) \\ \text{where} \;\; g(\tau, H_i) & \equiv & \displaystyle\sum_{p \in H_i} w_p e(p\tau)^2 \end{array}$$

Orthogonal to the choice of *locally-better* or one of the instances of *globally-better*, we can choose an appropriate error function for the constraints. *Locally-predicate-better* is *locally-better* using the trivial error



a) Initial state
b) Least squares, both p1 and p2 move
c) Only p2 moves
d) Only p1 moves
e) Undesirable: p1.x and p2.y move

Figure 1: Illustrations b) through d) are plausible solutions, while e) is undesirable

function that returns 0 if the constraint is satisfied and 1 if it is not. *Locally-metric-better*—the comparator of primary interest in this paper—is *locally-better* using a domain metric in computing the constraint errors. *Weighted-sum-predicate-better*, *weighted-sum-metric-better*, and so forth, are all defined analogously.

(Reference [5] also defines a third global comparator *worst-case-better*, which is not used in this paper.)

3 The Problem

There is a problem that arises for locally-metric-better when constraining compound objects such as 2-d points, which we call the *split stay problem.* Given a choice, for psychological plausibility the solver should pick a solution that satisfies the stays on both the x and y components of a given point, rather than on the x component of one point and the y component of another.

Consider a line with endpoints p1 and p2 and a midpoint m. There are constraints $(\mathsf{p1.x}+\mathsf{p2.x})/2 = \mathsf{m.x}$ and $(\mathsf{p1.y}+\mathsf{p2.y})/2 = \mathsf{m.y}$, and weak stay constraints on p1 and p2. Suppose we are editing m. In the new state of the line, we cannot satisfy the stay constraints on both p1 and p2.

Figure 1 illustrates. Figure 1a is the initial configuration. From the user's point of view, the plausible solutions include:

- minimize the squares of the errors for the stay constraints (Figure 1b)
- completely satisfy the stays on both p1.x and p1.y, and not on p2.x and p2.y (Figure 1c)
- completely satisfy the stays on both p2.x and p2.y, and not on p1.x and p1.y (Figure 1d)

These are all valid locally-metric-better solutions. A solution that is also valid but *not* psychologically plausible is to satisfy the stay constraints on the x part of one point and the y part of the other, for example, p1.y and p2.x, as shown in Figure 1e. In other words, in this solution we have split the stay constraints on p1 and p2. (This claim has been verified empirically—in an early implementation of one of our constraint satisfaction algorithms this happened, and indeed it looked strange.)

To see that these are all valid locally-metric-better solutions, in particular, for example, that 1e is not dominated by 1c, note that 1e has a smaller error for the stay constraint on p2.x than does 1c. One can similarly show that there is *no* other solution to the required constraints that is locally-metric-better than 1e.

4 An Improved Comparator

We now present a new comparator with the dubious name of *tilted-locally-metric-better* (TLMB) that provides a way to handle this situation.[1] The set of TLMB solutions can be defined by taking a given hierarchy, forming all possible hierarchies by breaking strength ties in all possible ways to form a totally ordered set of constraints, and taking the union of the sets of solutions to each of these totally ordered hierarchies.

[1]A preliminary description of this comparator is given in a technical report [2].

For example, consider the two constraints weak $x = 0$ and weak $x = 10$. The set of locally-metric-better solutions is the infinite set of mappings from x to each number in [0...10]. The TLMB solutions are defined by producing all the totally ordered hierarchies and taking the union of their solutions. In this case the two possible total orderings are:

weak $x = 0$, slightly_weaker $x = 10$
slightly_weaker $x = 0$, weak $x = 10$

These have solutions $\{x \mapsto 0\}$ and $\{x \mapsto 10\}$ respectively, so the set of TLMB solutions to the original hierarchy is $\{\{x \mapsto 0\}, \{x \mapsto 10\}\}$. As an aside, we hypothesize that the only psychologically plausible solutions to the example are $\{x \mapsto 0\}$, $\{x \mapsto 10\}$, and $\{x \mapsto 5\}$ (the least-squares solution), but not, for example, $\{x \mapsto 3.8\}$.

Next, we introduce a notion of a *compound constraint*, which is a conjunction of primitive constraints. For compound constraints, when we break the strength ties in defining the set of tilted-locally-metric-better solutions, we insist on mapping each primitive constraint in a given compound constraint to an adjacent strength.

Now, to define the desired solutions for Figure 1, we make each point stay be a compound constraint, so that the stays on p1 and p2 are:

weak (*stay* p1.x & *stay* p1.y)
weak (*stay* p2.x & *stay* p2.y)

In defining the set of tilted-locally-metric-better solutions, the total orderings of these constraints that we will consider have the stays on p1.x and p1.y both stronger than those on p2.x and p2.y, or both weaker. Thus, 1c and 1d are the only tilted-locally-metric-better solutions—so the undesirable solution 1e is ruled out. (The least-squares solution 1b is ruled out as well, which we regard as acceptable; if this solution is desired one should use the least-squares comparator.)

It is not sufficient just to define a notion of "compound constraint" without adding the notion of tilting—otherwise we would just sum the errors of the primitive constraints, which would allow us to trade off the errors arbitrarily; we could still satisfy the stay on the x component of one point and the y component of another (and thus admit solution 1e).

5 Relation to Other Comparators

The split stay problem does not arise for the least-squares-better comparator—to minimize the sum of the squares of the errors, normally the error will be distributed among all of the stay constraints on a collection of points (as in Figure 1b).

The problem can arise for the predicate comparators, but for those comparators a much simpler solution suffices—just use compound constraints, without tilting. (Since we just care whether or not a constraint is satisfied, and since a compound constraint is satisfied only if all of its primitive constraints are satisfied, the splitting problem does not then arise.)

Tilting is incompatible with the weighted-sum comparator. Using tilting to break a tie between two constraints with the same strength and weight could lead to incorrect results because one of the constraints would dominate a third constraint with the same strength and larger weight. However, the weights already can be used to break ties among constraints with the same strength—although there is the danger that a large number of constraints with small weights could overwhelm a constraint with a large weight.

6 Relation to Cassowary

As noted previously, Cassowary is an efficient incremental version of the simplex algorithm, tailored to the domain of interactive graphics. It finds a weighted-sum-better solution to a constraint hierarchy.

Our implementation of Cassowary favors solutions that satisfies some of the constraints completely, rather than ones that, for example, partially satisfy each of two conflicting equalities. (Thus, in Figure 1, it will find a solution that completely satisfies the stay on p1.x and not at all on p2.x, or vice versa, and not solution 1b.) These solutions are still legitimate weighted-sum-better solutions. Cassowary's behavior is analogous to that of the simplex algorithm, which always finds solutions at a vertex of the polytope even if all the solutions on an edge or face are equally good. (Of course, Cassowary behaves this way because the simplex algorithm does.)

However, without additional information, Cassowary may find Figure 1e rather than Figure 1c or Figure 1d. (As mentioned above, an early version of Cassowary actually did find such solutions, to the puzzlement of the user.)

This problem is addressed in our released version using a kludge, which works well in practice despite its dubious intellectual pedigree. One of the methods in the solver API is a method addPointStays(Vector points). The argument points is an array of points, whose x and y components are constrainable variables. This method adds a weak stay constraint to the x and y variables of each point. The weights for the x and y components of a given point are the same. However, the weights for successive points are each smaller than those for the previous point (1/2 of the previous weight). The effect of this is to encourage the solver to satisfy the stays on both the x and y of a given point rather than the x stay on one point and the y stay on another.

It was difficult to devise an example where it would give a bad answer—here is a contrived one. Suppose we have a line with endpoints p1 and p2 and a midpoint m. Suppose also we have constraints p2.x = 2*p3.x and p2.y = 2*p3.y. (This example is a bit strange since here we are using p3 as a distance from the origin rather than as a location—otherwise multiplying it by 2 is problematic.) If we give these points to addPointStays: in the order p1, p2, and p3, then the stays on p1 will have weight 1, those on p2 will have weight 0.5, and those on p3 will have weight 0.25. Then, a one legitimate weighted-sum-better solution would satisfy the stays on p1.x and p1.y, but another legitimate solution would satisfy the stays on p1.x, p2.y, and p3.y.

Since every weighted-sum-metric-better solution to a hierarchy is also a locally-metric-better solution [5], we could use Cassowary to find TLMB solutions. However, the obvious way to do this would involve introducing many more constraint strengths, which would lead to a less efficient implementation. Therefore, for now, we are continuing to use the decaying-weight technique provided by the addPointStays method—which, as noted above, works well in practice although could in theory allow a non-TLMB solution. Finding a completely principled and at the same time efficient solution remains an open problem.

Acknowledgments

This research has been funded in part by National Science Foundation Grant No. IIS-9975990, and for Greg Badros by an NSF Graduate Research Fellowship and the University of Washington Computer Science and Engineering Wilma Bradley fellowship.

References

[1] Greg Badros. *Extending Interactive Graphical Applications with Constraints.* PhD thesis, Department of Computer Science & Engineering, University of Washington, June 2000.

[2] Greg Badros and Alan Borning. The Cassowary linear arithmetic constraint solving algorithm: Interface and implementation. Technical Report 98-06-04, Dept. of Computer Science and Engineering, University of Washington, Seattle, WA, June 1998.

[3] Greg Badros, Alan Borning, Kim Marriott, and Peter Stuckey. Constraint cascading style sheets for the web. In *Proceedings of the 1999 ACM Symposium on User Interface Software and Technology*, pages 73–82, November 1999.

[4] Greg Badros, Jeffrey Nichols, and Alan Borning. SCWM—the scheme constraints window manager. In *Proceedings of the AAAI 2000 Spring Symposium Series on Smart Graphics*, March 2000.

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

[6] 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*, October 1997.

[7] Hiroshi Hosobe, Ken Miyashita, Shin Takahashi, Satoshi Matsuoka, and Akinori Yonezawa. Locally simultaneous constraint satisfaction. In *Proceedings of the 1994 Workshop on Principles and Practice of Constraint Programming*, pages 51–62. Springer-Verlag LLNCS 874, 1994.

[8] Håkon Wium Lie and Bert Bos. *Cascading Style Sheets: Designing for the Web.* Addison-Wesley, 1997.

[9] Michael Sannella. The SkyBlue constraint solver and its applications. In Vijay Saraswat and Pascal Van Hentenryck, editors, *Principles and Practice of Constraint Programming: The Newport Papers*, pages 385–406. MIT Press, 1995.

[10] Michael Sannella, John Maloney, Bjorn Freeman-Benson, and Alan Borning. Multi-way versus one-way constraints in user interfaces: Experience with the DeltaBlue algorithm. *Software—Practice and Experience*, 23(5):529–566, May 1993.

[11] Ivan Sutherland. *Sketchpad: A Man-Machine Graphical Communication System.* PhD thesis, Department of Electrical Engineering, MIT, January 1963.

[12] Brad Vander Zanden. An incremental algorithm for satisfying hierarchies of multi-way dataflow constraints. *ACM Transactions on Programming Languages and Systems*, 18(1):30–72, January 1996.