Constraints in Interactive Graphical Applications

Greg J. Badros
<gjb@cs.washington.edu>
Ph.D. General Examination
Department of Computer Science and Engineering
University of Washington, Box 352350
Seattle, WA 98195-2350

3 December 1998

Abstract

Constraints provide a declarative means for specifying relationships that we wish to hold true. Interactive graphical applications give rise to varying kinds of constraints, and researchers have developed diverse constraint solving techniques. I survey the classes of constraints used by numerous drawing, graph layout, visualization and animation systems. I describe a taxonomy of the constraint solving methods used to satisfy these systems and compare solver expressiveness and performance. Though backtracking algorithms have not yet been used successfully in interactive graphical applications, I summarize work on various backtracking algorithms and suggest ways to begin to improve their performance so they might be used in the user interface domain.

1 Introduction

From the inception of graphical user interfaces, systems have tried to use constraints to maintain relationships among on-screen entities [Sut63]. Constraints permit the designers or users of a system to express what they wish to hold true, rather than detail how to maintain the desired invariants procedurally. The fundamental strength of using constraints is this declarative specification of desired relationships. Constraints are especially natural for managing geometric systems including drawing, graph layout, visualization and animation.

Since any constraint system is limited by the expressibility and performance of its underlying constraint solver, increasing the power of solvers is a popular research area. There is substantial tension between the expressiveness of the constraints a solver can manage and its efficiency in finding a solution. Because of this fragile balance, system implementors typically hand-tune the tradeoff for each specific application involving constraints.

Numerous interactive graphical systems, including drawing, graph layout, visualization and animation systems, embed a constraint solver to manage the geometric layout of on-screen objects. These interactive and geometrically-based systems used in user interfaces make stringent demands of the constraint solving technology—the solver must be powerful enough to handle geometric constraints, fast enough for real-time interaction, and predictable enough to not confuse the user.

Section 2 of this paper discusses the classes of constraints occurring in various interactive graphical applications. I then explain and categorize the supporting constraint-solving technologies

with respect to the kinds of constraints they support. Section 3 details this taxonomy, and compares the expressiveness and performance of the solvers.

One class of constraints that no interactive graphical application yet attempts to solve is disjunctions. Batch constraint solvers (e.g., Prolog, CLP($\Re$) [JMSY92]) employ back-tracking as a means of exploring the space of alternatives. Section 4 summarizes the findings of an important paper analyzing backtracking algorithms. Section 5 discusses re-targeting these algorithms' for an interactive constraint solver that permits disjunctions. Finally, Section 6 concludes by summarizing important problems that future work must address.

2 Constraints across applications

Interactive graphical applications use constraints primarily to express desires in geometric layout. All of these systems are similar in that they expose constraints to the end user. However, the representations of constraints and to what they are attached vary substantially. Representation can influence the expressiveness and efficiency of a constraint solver by altering what kinds of relationships can be specified within the limitations of the solver.

For example, if line segments are represented using a starting point, direction, and length, a constraint that two lines are the same length or parallel is easy to express using a simple linear equality. Those same relationships will require non-linear equations if line segments are stored as coordinates of the start and end of the segment. Since solving general non-linear constraints is computationally difficult, systems often rely on domain-specific methods to handle the non-linearities they deem most important.

I now examine constraints in applications for drawing, graph layout, visualization and animation systems. For each application, I highlight the unique and interesting features of the system while listing the class of constraints it handles. For an overview of the kinds of constraints supported by each system, the solving technique employed, and the performance of the solver, see Table 1. Most notably, this section will show that applications tend to pollute their use of constraints based on limitations of the underlying solver. Comparing constraint expressiveness is more meaningful relative to the satisfaction algorithms, rather than the application domains. That discussion and details of each system's particular solving algorithms are deferred until Section 3.

2.1 Drawing

Interactive drawing applications employing direct-manipulation [Sch83] techniques have been very successful. For professional-looking diagrams and illustrations, however, they sometimes fall short due to the lack of precision in the diagram. Most conventional drawing systems permit alignment of objects, but such relationships are only enforced once—at the time the command is issued. If an aligned object is later moved, the other objects do not follow. Constraint-based drawing systems permit the persistence and maintenance of desired relationships to ease editing burden and ensure precision in the diagram. To date, drawing programs are the most common interactive graphical applications to use constraints.

Sutherland's Sketchpad, the earliest interactive constraint-based system, permits constraints to be explicitly specified about the objects in the figure. For example, a pair of lines can be made equal in length, or an angle can be marked as a right angle [Sut63, App. A]. Additionally, the user implicitly adds constraints through the use of the "pseudo-pen location" which locks the input pointer position onto topologically-important locations in the diagram. A similar input technique called "snap-dragging," developed years later by Bier, uses the snapped-to positions only for their

	System	Author (Year)	Constraints supported	Solving technique	Performance
Drawing	Sketchpad	Sutherland (1963)	geometric	LP, relaxation	$O(n)$, $O(n^2)$
	IDEAL	Van Wyk (1982)	geometric in complex plane	LP w/o planning	$O(n^2)$
	Juno	Nelson (1985)	CONG, PARA, HOR, VER	iterative numeric	$O(n^3)$
	Juno-2	Heydon and Nelson (1994)	CONG, PARA, HOR, VER	optimized iter. num.	$O(n^3)$
	Briar	Gleicher and Witkin (1994)	points-on-object,coincident	differential methods	$O(n^3)$
	Unidraw	Helm et al. (1995)	linear (in)equalities	direct numeric (QOCA)	$O(n^3)$, $O(n^2)$
	GCE	Kramer (1992)	geometric	DOF analysis	$O(ng)$, $O(n \log n)$
	Chimera	Kurlander (1991)	geometric	symbolic, numeric	$O(n^2)$
	Pegasus	Igarashi et al. (1997)	geometric	CLP(R)-like	$O(2^n)$
Graph	GLIDE	Ryall et al. (1997)	VOFs	spring simulation	polynomial
	CGL	He et al. (1996)	linear (in)equalities	iter. numeric	$>$ 1 sec (tree $n = 16$)
Visualization	TRIP*N*, IMAGE	Takahashi et al. (1998)	linear geometric	graph-layout, direct & iterative	"needs to be faster"
	ICOLA	Oster & Kusalik (1998)	linear inequalities	extreme-bound propagation	$O(n + v)$
	Penguins	Chok & Marriott (1998)	linear (in)equalities	direct numeric (QOCA)	interactive ($n \leq 700$)
Animation	TLCC	Gleicher & Witkin (1992)	geometric (on camera image)	differential methods	$O(n^3)$, non-interactive
	Animus	Duisberg (1987)	arbitrary acyclic	LP, relaxation	$O(n)$, $O(n^2)$
	JIM, Parcon	Griebel et al. (1996)	linear (in)equalities, geometric	iterative numeric	$<$ 1 sec ($n \leq 100$)

Table 1: Overview of constraints and solvers in interactive graphical applications.

absolute location [BS86]. Sketchpad, in contrast, stores and maintains the constraint relationships as objects are rearranged.

Sketchpad was ahead of its time; IDEAL, the next constraint-based system specifically targeting drawing, appeared almost twenty years later [VW82]. Unlike Sketchpad, IDEAL is strictly a textual language for specifying pictures—it is not an interactive system. IDEAL permits specifying arbitrary non-simultaneous constraints on points in the complex plane. The drawing is then created procedurally from a configuration of the points that satisfies the constraints.

Like Sketchpad, Juno and Juno-2 [Nel85, HN94] are interactive systems. Juno permits specifying constraints on points and line segments. There are only four predicates: `HOR` and `VER` express the horizontal and vertical relationship between pairs of points, while `CONG` and `PARA` are congruence and parallel relationships (both non-linear) between pairs of line segments. Juno's constraint relationships are specified at a higher level of abstraction than Sketchpad or IDEAL, though internally they are maintained as numerical mathematical relationships. Juno provides double-view editing, where both the graphical picture and the (partially declarative) program that constructed it are viewed simultaneously. Interactive direct-manipulation of the picture is reflected immediately as implicit edits of the program's text.

Briar [GW94] is an interactive drawing editor that permits expressing exactly two geometric constraints: *points-on-object* and *points-coincident*. Though this set of relations is limited, Briar re-gains expressive power by allowing "alignment objects." Such objects exist only as constraint-assistance artifacts and are not part of the final drawing (e.g., alignment objects are not output when the figure is printed). For example, the constraint that point p is distance k away from point q can be expressed by placing an alignment circle centered at point q, and constraining point p to be on that circle. Constraints among both regular and alignment objects are specified implicitly through an extension of Bier's snap-dragging suitably named "augmented snap-dragging" [Gle92]. Adding a constraint on a new object corresponds directly to creating that new object while the pointer is snapped onto a pre-existing object or point. Unlike other systems, in Briar there is never a need to manage constraints explicitly. Removing a constraint is performed by breaking constraints through "ripping apart" objects—the user specifies only the desired effects and the system chooses which constraints must be eliminated.

Unidraw [HHMV95] is an extension of an earlier direct-manipulation drawing program. It permits arbitrary simultaneous linear equalities and inequalities among attributes of its various predefined objects. This class of constraints is shared by the CDA [Not98] drawing application, and Penguins [CM98], a drawing-editor construction framework (analogous to YACC for generating language parsers).[1] Unidraw is the only drawing editor without any support for non-linear constraints. Also, Unidraw is unique among constraint-based drawing editors in its support for `undo` and `redo` operations. Though typically challenging to implement, these features are permitted by Unidraw's ability to easily enable and disable constraints and to save and restore the state of the entire constraint system.

Kramer's Geometric Constraint Engine [Kra92] (GCE is an extension of his earlier The Linkage Assistant, or TLA) is not specifically a drawing editor, but solves the same class of geometric layout problems. GCE permits five classes of binary constraints between geometric objects, or *geoms*: distance between a point and a point, line, or plane; distance between line and a circle; and angle between a pair of vectors. These constraints are tied to the geometric degrees-of-freedom analysis performed in Kramer's underlying solver (see Section 3.5).

Chimera [KF91] not only supports drawing constrained figures, but also provides a constraint inference engine. Kurlander's system permits the constraints shown in Table 2. Like GCE, the

[1] See Section 2.3 for more discussion of Penguins.

constraints supported by Chimera are directly related to a solving technique characterized by reasoning about transformational groups. Chimera's inference engine works by comparing multiple snapshots and constraining the things that are invariant (within a tolerance) across the snapshots. Instead of explicitly stating what relationships one wants to hold, the user must vary all of the degrees of freedom that are meant *not* to be constrained. The invariant-detection tolerance mechanism employed by Chimera for detecting invariants is similar to Pavlidis's automatic beautification in PED [PVW85]. PED, however, only infers the constraints and makes the diagram more precise, whereas Chimera dynamically and interactively maintains the constraints.

Absolute Constraints	**Relative Constraints**
Fixed vertex location	Coincident vertices
Distance between two vertices	Relative distance between pairs of vertices
Distance between parallel lines	Relative distance between pairs of parallel lines
Slope between two vertices	Relative slope between two pairs of vertices
Angle between three vertices	Equal angles between two pairs of three vertices

Table 2: Constraints permitted by Kurlander's Chimera [KF91, p. 14]

Pegasus (Perceptually Enhanced Geometric Assistance Satisfies US) [IMKT97] is a rapid sketching tool that also interactively infers constraints. Pegasus recognizes seven kinds of constraints: connection, parallelism, perpendicularity, alignment, congruence, symmetry, and interval equality. Unlike Chimera, the constraints Pegasus infers are not maintained (i.e., they are one-shot corrections, more akin to snap-dragging).

2.2 Graph layout

Graph layout is a particularly challenging application for use of constraints. The aesthetic criteria by which a graph layout is judged is difficult to express using simple relationships. In general, graph layout requires minimization of a non-quadratic objective for visually-pleasing results. Optimization criteria often includes eliminating node overlaps, minimizing edge crossings, and maximizing symmetries. Classical graph layout algorithms are expensive, batch-oriented computations [DBETT94, DBETT99].

Wieqing He and Kim Marriott describe a non-interactive system for constrained graph layout where the constraints are used to further specify requirements above and beyond a classical batch layout algorithm's aesthetic criteria [HM96, MCF98]. They use three different layout modules and augment them with a constraint solver to enforce the user-specified simultaneous linear equality and inequality constraints.

GLIDE [RMS97] is an interactive system for graph layout which uses constraints in the form of Visual Organization Features (VOFs). The VOFs it handles (inherited from earlier work by Marks) include alignment, even spacing, sequence, cluster, T-shape, zone, symmetry, and hub-shape [DFM93, KMS94]. Phantom nodes (similar to Gleicher's alignment objects) are used as alignment guides. All of these constraints specify local relationships among small groups of nodes.

Other interactive graph layout systems do not include any general constraints but simply provide an interactive means of viewing and manipulating constraints laid out through conventional algorithms [HH91, Hen92].

2.3 Visualization

Visualization systems provide pictures for abstract data. These visual representations permit viewers to exploit their perceptual skills in exploring data. Graph layout (see Section 2.2) is one well-studied domain of visualization. Interactive visualization systems can use constraints to aid in producing semantically meaningful pictures.

TRIP (TRanslate Into Pictures) [KK91] and its successors TRIP2, TRIP2a, TRIP3D, TRIP3, and IMAGE [TMM+98] are all frameworks for visualizing abstract data. The TRIP systems provide mapping rules to translate between an Abstract Structure Representation (ASR)[2] and a Visual Structure Representation (VSR). The VSR level includes graphical objects along with geometric constraints. Some constraints span multiple objects: horizontal/vertical, spacing, and averaging; another constraint specifies the position of objects (the *at* constraint). Finally, there are graph-layout constraints for adjacency and for drawing edges to connect two nodes in the VSR. From the VSR, a picture representation (PR) is generated by solving the constraints (using the COnstraint-based Object Layout system, or COOL). Constrained editing of the resulting picture is not be permitted. The TRIP systems' constraints are very similar to those provided by the GLIDE graph layout system (see Section 2.2).

The Wand visualization system embeds ICOLA (Incremental Constraint-based Object Layout Algorithm) [OK98]. Wand's architecture is similar to TRIP, though it specifically targets visualization of logic program execution. ICOLA provides only linear inequality constraints—there is no way to enforce that two object attributes are equal. This inability to maintain equality constraints is unique among the systems surveyed. ICOLA's constraint language allows higher-level, "aliased," constraints which map into one or more of four basic constraints: `left_of`, `horizontal_distance`, `above`, and `vertical_distance`. The fifth basic constraint, `connected`, draws an arc or edge between two objects (as did the similar procedural "connects" constraint in TRIP). The DOODLE (Draw an Object-Oriented Database LanguagE) [Cru95] system provides similar functionality and constraints but provides a visual rather than textual specification language.

Penguins [CM98] is an intelligent diagram editor construction toolkit. It is to drawing editors what YACC is to parsers. The Penguins system uses constraints in two separate ways. First, it uses them for visual parsing, using the theory of constraint multi-set grammars (CMGs) [Mar94, CM95]. After building an internal abstract representation of a picture, editors created using Penguins then permit direct manipulation of the picture while interactively maintaining the constraints. Penguins-generated drawing editors permit arbitrary linear equality and inequality constraints.

2.4 Animation

Animation, like graph layout, is an especially challenging domain for the application of constraints. Much of the work on using constraints with animation systems is for non-interactive solvers. Constraint-based motion adaptation [GL96], space-time constraints [WK88], and motion interpolation methods [Bro88] all address solving huge multi-frame animation systems over time to provide meaningful character or object movement subject to certain desires. These are batch systems whose computation expense is justified in light of the resources required for subsequently rendering the frames of the animation.

Numerous visualization systems, including TRIP (see Section 2.3), and widget toolkits (see Section 2.5) provide animations meant to provide user-feedback for global changes made to the visual state of the system. TRIP provides "transition mapping rules" which are abstractions of procedures for interpolating between visual representations of two states. Artkit [HS93] provides a

[2]The ASR, in turn, is derived from an Application Representation (AR).

similar "transition" abstraction for animations to be used when an object's state changes. Amulet [MMMF96] exploits the constraint solving framework's monitors guarding assignments to slots (i.e., the ability to execute code on every assignment) to provide a similar interpolated animation when a slot's value is set.

Animus [Dui88, BD86] uses the ThingLab system [Bor79] and provides animations for its simulations using constraints on time. In Animus, time is treated as a distinguished global variable. Animus provides two time-related constraints: 1) time function constraints which act as a declarative specification of events and responses to those events (similar to the Amulet monitors mechanism, above); and 2) ordinary differential equations for describing continuous motion (similar to Briar's use of differential methods [GW94], see Section 3.5).

Griebel et al. undertook a similar use of constraints for animation within the Pictorial Janus (PJ) visual programming language [GLM+96]. Other constraints they provide include linear equalities and inequalities, product equalities and inequalities, point coincidence, and distance constraints. They also provide a `c-disjoint` (circular-disjoint) relationship to prevent objects from overlapping.

2.5 Other interactive graphical application domains

Other application domains have used geometric constraints with some success. Window layout systems employing constraints include the Constraint Window System (CWS) [EL88] for Smalltalk, a constraint-based tiled window manager called RTL/CRTL (Research and Technology Laboratories Constrained Rectangular Tiled Layout) [CSI86] for the Sapphire Window System, and SCWM (Scheme Constraints Window Manager) [BS98] for the X11 window system. These systems permit specification of constraints over the windows regarding their presence, size and location, adjacency and alignment, and hierarchical organization. All systems restrict their constraints to rectangular windows, but face stiff challenges due to the highly dynamic nature of windowing environments where new objects come and go frequently.[3]

A similar application is web page layout. A prototype Java-based web browser permits page layout and applet layout to be specified using linear equalities and inequalities [BLM97, MCF98]. For page layout, only rectangular bounding boxes are considered.[4] The browser interactively lays out the page again and again as the enclosing window size changes, preserving the desired constraints.

User interface widget toolkits are second only to drawing editors in their aggressiveness using constraints. Numerous widget toolkits including Amulet [MM95, MMM+97], its predecessor Garnet [MGD+90a], and OPUS of the Penguims[5] user-interface management system [HM90] all provide one-way constraint solvers for relating the components in a widget hierarchy. Bramble [Gle93] is the toolkit with which the Briar (see Section 2.1) drawing editor is implemented.

Other constraint-based interactive systems have been used for graphical search and replace [KF92], curve manipulation independent of representation [FB93], and colour management for windowing interfaces [Mac91].

[3] Vander Zanden et al. discuss ways to cope with the dynamic relationships maintained by windowing systems such as SCWM that may be worth integrating into its solvers [VZMGS]. That work is a generalization of Borning's "paths" [Bor79, p. 39–41].

[4] Some work has been done to extend the Cascading Style Sheets level 2 (CSS2) specification of box layout to use constraints [Mic98].

[5] Not to be confused with Chok and Marriot's Penguins intelligent diagram editor toolkit.

2.6 Summary of application domains

As Table 1 shows, constraints used by different systems within an application domain are generally not especially closely related. The similarities that do exist result more from the underlying solver than from the needs of a particular class of applications (see the following section). Another dimension along which the applications varied is the level of abstraction that the constraints are managed at.

Systems including Briar [GW94], GCE [Kra92], Chimera [KF91], Pegasus [IMKT97], and GLIDE [RMS97] express constraints on complete objects in the system. These constrain, e.g., angles between vectors, Euclidean distances between points and lines, coincidence of a point and an object, or symmetries. Such constraints are the highest level of abstraction provided by any of the systems considered.

Several drawing systems, such as IDEAL [VW82], Juno and Juno-2 [Nel85, HN94], permit specifying constraints on points, and then parameterize drawings based on the locations of those points. After the constraint satisfaction algorithm solves for absolute point locations, procedural (i.e., not declarative) code fragments connect lines, draw circles, and otherwise flesh out the drawing. These systems provide greater flexibility in final appearance, but expose a mixed declarative and procedural interface to the end user. (A similar mix of paradigms is used by Animus [Dui88], which is built on ThingLab [Bor79].)

A third general approach, used by Unidraw [HHMV95], CDA [Not98] and Penguins [CM98, MCF98], involves expressing numerical constraints on attributes (also called reference points, selectors, aspects, and landmarks) of objects which have an implicit visual representation. The modification of a constrained attribute's value (e.g., a rectangle's northwest corner, or a circle's center) is reflected by updating the position of the corresponding on-screen object. "Internal" constraints often implicitly relate attributes to each other, e.g., relating two corners of a box: box.ne.x = box.nw.x + box.width.

The level of abstraction for specifying constraint relationship is significant because it can decouple the application from the solver. Higher levels of abstractions may rely less on solver dependencies, and permit substituting out a more efficient or powerful solver without influencing the rest of the system. Limiting the constraints to simple linear numerical constraints may provide a similar benefit, despite being at the opposite end of the abstraction level spectrum.

3 Interactive satisfaction algorithms

The preceding section demonstrates that the constraint types provided by specific genres of applications are not especially consistent. Different drawing programs, though attacking the same problem, have different formulations of the relationships they provide: Gleicher's Briar [GW94], Kurlander's Chimera [KF92], Nelson's Juno [Nel85], and Unidraw [HHMV95] all provide different constraint mechanisms. Conversely, Unidraw, Penguins [CM98], and Scwm [BS98] all expose the same constraint-solving interface, yet they belong to different application domains. The constraints that a specific application permits the user to specify are dependent not on the kind of application but on the underlying solving technology.

Juno provides an excellent example of the correlation between constraints permitted and the underlying solving technology. Juno's author, Greg Nelson, explains that he had attempted to provide a fifth predicate, `CC` (counter-clockwise), to disambiguate under-constrained systems. However, because the counter-clockwise relationship translates into an inequality constraint which the Newton-Rhapson solver Juno uses could not easily handle, he discarded that approach and instead chose to exploit a "feature" of Juno's underlying iterative numerical solver: the solution's depen-

dence on the initial guess. Thus, Nelson added the ability to provide hints to the solver (which in turn led to the need for `REL` construct) [Nel85, p. 238–239].

Though ideally we would like to consider what constraint relationships our application needs and provide exactly those capabilities, it is clear that satisfaction algorithms influence the system's design. This section discusses the various satisfaction algorithms developed for interactive geometric applications, and relates them to one another while pointing out their strengths and weaknesses. Figure 1 graphically depicts the relationships among the over twenty different constraint satisfaction algorithms considered here.

3.1 Common issues

The issues constraint solvers must address and some of the approaches for remaining efficient are similar. All constraint systems must deal with under-constrained systems. An under-constrained system has remaining degrees of freedom remain so multiple possible solutions exist. Constraint hierarchies [BMMW89, FBWB92] provide a popular and well-studied means of removing ambiguity by over-constraining the system with constraints at decreasing levels of preference. Then a simple greedy algorithm permits using just enough constraints to maintain a fully-specified system and a unique solution. An alternative technique for choosing a solution from many possibilities is to use an objective optimization function to rank assignments, and choose the solution with the best score.

A related concern for solvers is to maintain spatial stability of the system. When a geometric system is under-constrained, successive solutions are more useful if they are sufficiently similar to prior configurations. A satisfaction technique that alternates between two (visually) distantly related solutions will be confusing to the end user. Supporting the "principle of least astonishment" [GLM+96] is a ubiquitous goal for constraint satisfaction algorithms. "Stay" constraints are used in solvers supporting constraint hierarchies to express the desire for things to remain where they are unless some stronger constraint forces them to move. Numeric solvers often disambiguate under-constrained systems and provide spatial stability by minimizing change from the previous solution.

Another commonality among the solver implementations is that they exploit sharing of data structures to provide the equality or coincidence-of-points constraint. Many systems manage their data structures to alias related variables when such a constraint is added and to "explode" the variables back into their unrelated instances when such a constraint is removed. This technique is largely independent of the satisfaction algorithm itself and often provides a substantial performance improvement by reducing the size of the system (equality constraints are especially common).

The variable aliasing optimization is generally beneficial because it performs work outside of the constraint system. Other techniques external to the solver are used to increase expressiveness. For example, the `connects` relationship (remember, this states that two objects in a diagram should be connected by a line) for graph drawing and visualization systems is often not maintained by adding constraints to compute appropriate positions for the edge endpoints. Instead procedural code simply draws the requested edge, performing its own computations as necessary. At first glance, this technique seems to defeat the beneficial declarative nature of constraints. However, since the constraint solver is still responsible for ensuring that the relationship holds, it does not matter to the end user whether the main constraint system is the engine that satisfies the relationship. These special relationships, by necessity, must be disconnected from the rest of the constraint graph, and the technique can be seen as simply a very restricted domain-specific sub-solver, similar to the sub-solvers used by Ultraviolet [BFB98] or DETAIL [HMT+94].[6]

[6] In contrast, the Juno systems [Nel85, HN94] take this approach to an extreme and permit the user to specify

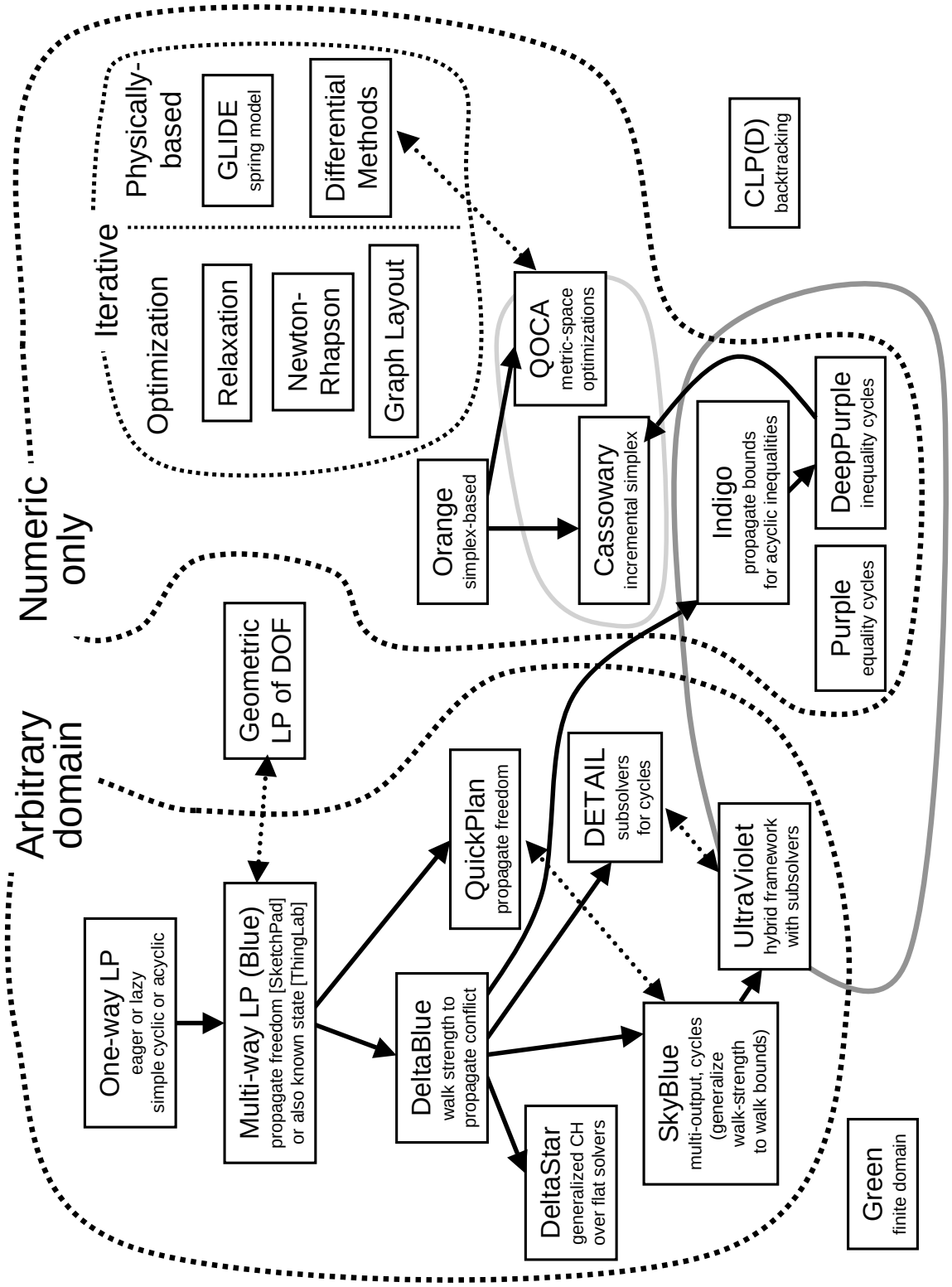

Figure 1: Taxonomy of interactive constraint solvers. General classes of algorithms are demarcated by dotted lines, containment of sub-solvers by light solid lines, arrows indicate evolving relationships, and proximity roughly correlates with relatedness. Especially closely-related but independently designed systems are connected by bi-directional dotted arrows.

Interactive constraint solvers are often split into a planning, or compilation, stage and an execution stage. During planning, the solver pre-computes all state that will remain fixed throughout a class of executions. These restrictions permit the system to be more efficient during the corresponding solver iterations.[7] The basic idea is similar to loop-invariant code motion and dynamic compilation techniques. Some time is spent in advance, when it is less precious, to increase the performance during the tight interaction and animation loop.

The remainder of this section discusses the several constraint satisfaction algorithms and considers their performance and expressiveness.

3.2 Local-propagation based solvers

Local propagation (LP) is one of the earliest-developed constraint solving techniques and is conceptually very simple. Sutherland's initial formulation of local propagation, the "one-pass method," [Sut63, p. 58–59] is a highly efficient algorithm used whenever possible before falling back to his more general (but slower) relaxation algorithm (see Section 3.3). The most significant limitation of propagation-based solving is their inability to consider more than one constraint at the same time. This prevents solving simultaneous linear equations and other systems which require manipulations of multiple constraints at once. Such simultaneous interactions among constraints appear as cycles in the constraint graph.

Local propagation techniques vary along several dimensions: one-way vs. multi-way; constraint hierarchies vs. flat systems; acyclic vs. cycles allowed; single-output vs. multiple-output; and equality (functional) relationships only vs. inequalities permitted. See Table 3 for an overview of the systems described in this section.

3.2.1 One-way LP constraint solvers

The simplest local propagation solvers are embedded in widget layout kits such as ARTKit's Penguims [HM90], Amulet [MMM+97] and Garnet [MGD+90a]. These perform only one-way solving—a constraint such as $x = y + z + 10$ will be maintained only by setting x (the output variable) and never by setting y or z (the input variables). Though this example constraint is numeric, one of local propagation's strengths is that the relationships may be specified over an arbitrary domain—the only restriction is that the output value is determined by a function (e.g., inequality constraints are non-functional and require a more powerful propagation algorithm).

Since one-way constraints are always maintained by evaluating the same assignment method, the satisfaction algorithm must simply decide which constraints' methods must be invoked and in what order. Consider the example in Figure 2. The corresponding constraint graph with variables as nodes and directed (because we are discussing one-way solvers) multi-edges representing constraints appears in Figure 3. After a variable is changed, all downstream variables must be updated by enforcing the constraints in topological order.[8] The one-way LP solver propagates values along the constraint graph.

Thus, simple one-way constraint solvers can maintain their relationships using a standard

arbitrary code parameterized on the points. Here the benefits of declarative specification are largely lost to provide greater flexibility in drawing.

[7] In some cases, solvers using dynamic languages actually compile the code of the inner loop.

[8] Alternatively, downstream variables may be marked invalid, and the constraints can be lazily enforced when a variable's value is requested. Experience suggests that for common layout tasks the cost in maintaining the invalid bit exceeds the savings from unused evaluations [MGD+90b].

Solver	**Multi-way?**	**C.H.?**	**Cycles ok?**	**Multi-output?**	**Ineqs.?**
Sketchpad	yes	no	no	no	no
ThingLab	yes	no[a]	no	no	no
ARTKit Penguims	no	no	no	no	no
Garnet and Amulet	no	no	partially	no	no
(Delta)Blue	yes	yes	no	no	no
QuickPlan	yes	yes	yes	yes	no
SkyBlue	yes	yes	yes	yes	no
DETAIL	yes	yes	yes	yes	no
Indigo	yes	yes	no	no	yes
Ultraviolet[b]	yes	yes	yes	yes	yes

[a] In early work, Borning called these meta-constraints; he later integrated them into subsequent simulation environments [Bor79, p. 94].

[b] Ultraviolet is actually a meta-solver that is responsible for graph partitioning and invoking sub-solvers. This chart reflects the capabilities of the various sub-solvers it embeds

Table 3: Overview of local propagation algorithms.

$$
\begin{array}{lllll}
C_1: & m & = & \frac{(x_1+x_2)}{2} \\
C_2: & x_1 & = & \text{pointer position} \\
C_3: & x_2 & = & x_1 + 6 \\
C_4: & r & = & m^2
\end{array}
$$

Figure 2: Simple set of constraints for local propagation examples.

topological sort, based on a depth-first search of the directed constraint graph.[9] Its computational complexity is $O(V + C)$, where V is the number of variables (i.e., nodes), and C is the number of constraints (i.e., edges). Though the structure of the constraint graph only changes when constraints are added or removed, the values propagated can change rapidly. For example, when the user is interacting with the system shown in Figure 3, x_1 will vary as the user moves the mouse pointer. LP solvers optimize for this by maintaining the topologically sorted graph and simply traversing it while executing the methods for each new position. This reflects the previously-mentioned separation in planning (sorting the graph) and executing (firing the constraint-enforcing methods) which we will see again and again. Readers fluent with linear algebra may recognize the planning stage as the ordering of rows and the execution stage as the back-substitution phase in the solving of a system of equations using Gaussian elimination. However, remember that LP is

[9] This algorithm only works because we restrict the constraint graph to not contain cycles—more powerful techniques are required if constraints interact (see Section 3.4).

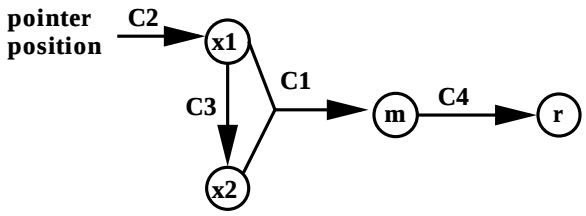

Figure 3: One-way (directed) constraint graph for Figure 2.

not limited to numeric domains—a constraint relationship can, for example, specify that a string, s, should always contain the printable form of the current color of a circle.

The separation of planning and execution is not essential, but is an optimization. Van Wyk's constraint satisfaction algorithm for IDEAL is a simple work-list approach which propagates state using the current constraint if enough variables are already assigned values, and otherwise delays that constraint by putting it back at the end of the work-list [VW82]. This worst-case $O(n^2)$ algorithm is an inefficient implementation of LP.

3.2.2 Multi-way constraints and solvers

One-way constraint solvers are exceptionally fast and easy to implement, but they largely sacrifice the declarative nature of constraints. Multi-way constraints are a generalization which permit the constraint solver more freedom in choosing how to satisfy a given constraint. Consider C_3 from Figure 2: $x_2 = x_1 + 6$. A one-way constraint solver may only change x_2 in response to changes in x_1, while a multi-way solver is free to set $x_1 \leftarrow x_2 - 6$ instead. Sketchpad [Sut63] and Borning's ThingLab [Bor79] are both multi-way LP solvers.

In ThingLab, constraints are specified by predicates and one or more satisfaction methods as in Figure 4. A multi-way LP algorithm not only has to choose the order by which to satisfy constraints, but also which method should be invoked for each. Figure 5 is the (now largely undirected) multi-way constraint graph that corresponds to Figure 3. Visually, the additional chore of the multi-way LP solver is to put arrowheads on each undirected edge. Not all edges are undirected—C_2, which constrains x_1 to the pointer position, can only be satisfied by changing x_1 so it remains represented as a directed edge.[10] The selection of edge directions corresponds to choosing a satisfaction method for each constraint. A solution to this planning stage assigns directions to all edges such that no variable node has two incoming edges—that would signify a conflict in that two constraints are competing to affect the same variable's value.

The earliest solving algorithm for multi-way constraint graphs, the aforementioned one-pass method, propagates freedom instead of values. Variables only constrained by a single relationship (i.e., those with only a single adjacent edge) are called "free" variables. These variables have enough degrees of freedom that they can be satisfied no matter what the assignments to the other variables are, so their assignment method is chosen to execute last. The edge is directed to select the method that assigns to the free variable, and that method is added to an execution list. Then the free variable node and planned-to-be-satisfied constraint edge are removed from the graph, and the process repeats. In this way, an execution plan is created in reverse order of ultimate execution [Sut63, pp. 58–59] [BD86, p. 363]. The propagation of values popularized by widget toolkits (see

[10] Even this restriction could be removed if the user's mouse had a motor so it could move around under program control!

$$
\begin{array}{lcll}
m & = & \frac{(x_1+x_2)}{2} & \\
 & m & \leftarrow & \frac{(x_1+x_2)}{2} \\
 & x_1 & \leftarrow & 2m - x_2 \\
 & x_2 & \leftarrow & 2m - x_1
\end{array}
$$

Figure 4: Predicate and three satisfaction methods for specification of multi-way constraint. In practice, for linear numeric constraints the satisfaction assignments can easily be inferred. For other domains where inverses are harder to compute, the methods may need to be explicitly programmed.

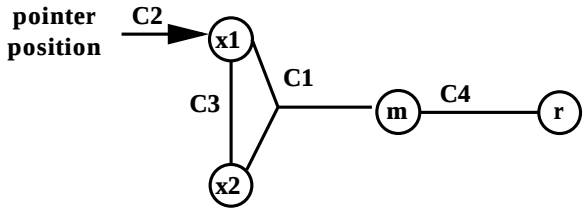

Figure 5: Multi-way constraint graph for Figure 2.

Section 3.2.1) is an extension introduced by Borning and originally called propagation of known states [Bor79, p. 67]. While propagation of freedom exploits nodes with *enough* degrees of freedom so they can assigned values last, propagation of known state proceeds towards a solution by finding nodes that have *no* degrees of freedom so they can be assigned values immediately.

With the extra expressiveness of multi-way constraints comes a substantial complication: multiple possible plans may exist to solve the same system. If we remove C_2 from Figure 2 there are two possible plans for executing as m is changed (see Figure 6). This ambiguity is not just an artifact of the solver, but is fundamental to the problem specification—it is under-constrained. ThingLab has the notion of meta-constraints which control aspects of the solver's behaviour. For example, the user textually orders the listing of the satisfaction methods to indicate which assignment should be performed when multiple possibilities exist. This type of meta-constraint was later refined into the now-classic notion of a constraint hierarchy [BMMW89, FBWB92] where constraints may be specified at multiple levels of preference.[11]

"Blue" is a multi-way LP solver that respects constraint hierarchies by finding the "best" solution [FBMB90]. Best is defined in terms of comparators. Blue uses the locally-predicate-better notion to compare two solutions and determine which is best. A locally-predicate-better solution satisfies all the required constraints and successively weaker constraints at least as well as its competing solutions, and satisfies at least one more constraint. For example, by the locally-predicate-better comparator, it is more desirable to have a solution that satisfies all required constraints and a single strong constraint rather than one that satisfies all the required constraints and ten (or a million) weak constraints. The comparator is "local" in that it compares solutions constraint by constraint, instead of computing some global measure of how satisfied all the con-

[11]The DeltaStar solver shown in Figure 1 was designed simply to aid research in constraint hierarchies by parameterizing a constraint-hierarchy by an arbitrary flat solver [FBWB92].

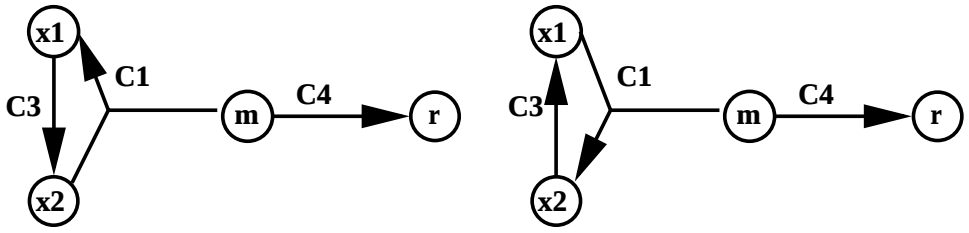

Figure 6: Two possible plans for executing Figure 5 as m changes.

straints are; the comparator is "predicate" in that all that matters is whether the constraint was satisfied or not, without regard to how closely the constraint is satisfied (i.e., the error). The locally-predicate better solution is designed to permit the use of a greedy algorithm for solving.

"DeltaBlue" is a suitably-named incremental version of the Blue algorithm. It maintains and incrementally updates a solution graph which represents a plan for recomputing variables' values to satisfy all satiable constraints in a constraint hierarchy subject to the locally-predicate-better comparator.

The key feature of DeltaBlue is its annotating of variable nodes in the method graph[12] with their "walkabout strength," or, more simply, walk-strength. The walk-strength of a variable is the weakest upstream constraint that could be un-enforced (i.e., removed or re-directed in the solution graph) to permit a different constraint to change the variable's value. Figure 7 shows a simple example [FBMB90, p. 58]. In particular, variable D's walk-strength is weak because constraint $C2$ is weak, thus denoting that DeltaBlue would only need to break a weak constraint in order to permit another (stronger) constraint to assign to D. Variable C's walk-strength is strong despite being the output of a required constraint because its input variable A's walk-strength is only strong; weaker walk-strengths propagate through stronger constraints.

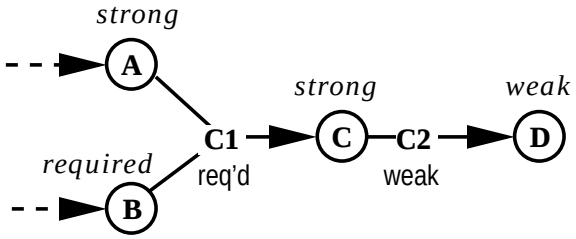

Figure 7: Example of walk-strength assignments to variables. Constraint strengths are below the constraint, current variable walk-strength assignments are in italics above the variable nodes.

Walk-strengths encapsulate the global knowledge needed to permit preserving locally-predicate-better solution plans across incremental constraint addition and removal. The key correlation between walk-strengths and solutions involves the notion of a *blocked constraint*—a constraint that

[12]Method graph is an alternative name for the constraint graph. Some authors use "constraint graph" to refer to the bi-partite graph with edges connecting constraints with the variables they constrain. In the bi-partite constraint graph, both constraints and variables are nodes. (e.g., see Figure 8).

is unsatisfied but has a strength stronger than the walk-strength of a potential output variable. The blocking constraint lemma states:

> If there are no blocked constraints, then the set of satisfied constraints represents a locally-predicate-better solution to the constraint hierarchy [FBMB90, p. 60]

This blocking lemma suggest the algorithm's strategy—the propagation of conflict. DeltaBlue's incremental maintenance of the method graph plan is straightforward [FBMB90, SMFBB93]. The algorithm's complexity remains $O(V+C)$ (as was simple LP). As mentioned before, assigning new values given the same configuration (i.e., execution) is especially fast (only $O(C)$ since at most one method is fired per constraint).

3.2.3 Extensible local-propagation solvers

There are three main limitations of DeltaBlue: 1) it can handle only functional constraints (e.g., it cannot manage inequalities); 2) it cannot solve cyclic constraint graphs; and 3) all methods must have exactly one output variable. The "Indigo" solver [BFB98] relaxes the first restriction by propagating bounds on value assignments instead of specific values—the bindings Indigo makes to variables are intervals. This generalization requires the solver to fire multiple interval tightening methods instead of just a single method performing a value assignment. Thus, if the constraints $a \leq 20$ and $a \geq 5$ are applied in that order, Indigo will first tighten a interval to $(-\infty, 20]$ and then to $[5, 20]$. These extra method invocations increase the complexity of the Indigo algorithm to $O(MC)$, where M is the maximum number of variables related by a constraint. The second and third restrictions are relaxed by the enhanced solvers described below.

SkyBlue [San94b] is a multi-way, multi-output solver that is capable of supporting sub-solvers for cyclic sub-graphs. Multi-output functions are useful for decomposing compound data structures and maintaining interacting constraints across multiple variables. The standard example is a two-input two-output constraint relating polar and Cartesian coordinates of a point. Support for multi-output functions is a necessary (though not sufficient) feature for a solver to support cycle-solvers.

As previously mentioned, cycles in the constraint graph correspond to simultaneous interactions of variables in the underlying problem. For example, the two constraints: $C_1 : x + y = 6$ and $C_2 : x - y = 2$ correspond to the bi-partite constraint graph in Figure 8. Because both constraints relate both variables, the graph is cyclic.[13] The primary shortcoming of all the LP solvers mentioned above is that they are able to reason about individual constraints only in isolation. When cycles appear in the constraint graph, more sophisticated algorithms must handle the more complex interactions. Another potential cause of cycles is the existence of redundant constraints—although such redundancies can often be eliminated by carefully analyzing the system, forcing the constraint specifier (often the end-user for interactive graphical applications) to avoid redundancies is unacceptable. Alternate views provide another approach to avoiding problems caused by circularities [Gos83, p. 27].

Cycles of linear numerical equality constraints correspond to systems of simultaneous linear equations which can be solved by such elementary algorithms as Gaussian elimination (see Section 3.4). The first challenge for the LP solver is in recognizing the cycles and invoking domain-specific sub-solvers on the connected subgraphs that LP is incapable of solving. As cycle-handling LP solvers find subgraphs with cycles, the solvers collapse those nodes into single meta-nodes and use the solution type of the enclosed constraints to assign a domain-specific sub-solver the task of

[13] Cycles in the bi-partite graph correspond to cycles in the method graph; in this case the method graph is just the two variable nodes connected by two distinct edges—one for each constraint.

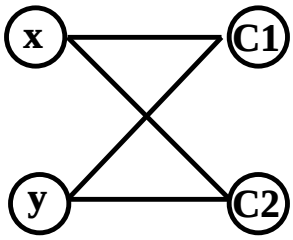

Figure 8: Bi-partite constraint graph showing constraints and the variables they relate.

assigning a valuation to the variables contained in the clumps. For the sub-solver to perform its task, it might need to assign values to multiple variables along the frontier where a collapsed meta-node interfaces with the full method graph. Thus, the main solver must permit multiple outputs for a single constraint (the aforementioned necessary but not sufficient condition for cycle-solving LP algorithms).

The SkyBlue solver's main contribution is the relaxation of the single-output restriction of DeltaBlue. In the presence of multi-output constraints, walk-strengths are no longer powerful enough to capture the relevant global information. The SkyBlue algorithm instead computes walkbounds—any strength equal to or weaker than the walk-strength—and maintains walkbounds incrementally as constraints are added and removed. The algorithm then computes the solution graph by building method vines using a backtracking algorithm [San94b]. Walkbounds and other optimization techniques help to reduce the needed backtracking substantially, but not completely. The backtracking makes SkyBlue's complexity exponential in the worst case.

QuickPlan [VZ96] is similar to SkyBlue but uses propagation of degrees of freedom (instead of propagation of conflict), searching for free variables and selecting methods to execute in reverse order. As it encounters conflicts planning its solution, it retracts the weakest strength constraint from the graph, saving it on a priority queue (ordered by strength). After the sequence of elimination and retraction steps, QuickPlan tries to re-add the retracted constraints in decreasing order of strength. The QuickPlan algorithm has $O(C^2)$ worst case complexity, it typically runs in linear time (recall that the single-output solver, DeltaBlue, is a linear-time algorithm).

DETAIL [HMT+94] is yet another multi-output cycle-solver-capable LP algorithm. Its algorithm is similar to the above, and it embeds three sub-solvers: one for locally-predicate-better constraints, one for least-squares-better linear equality systems, and one that uses a spring model (similar to GLIDE [RMS97]).

Ultraviolet, a meta-solver for invoking sub-solvers, first partitions the top-level constraint graph, and then solves the connected subgraphs independently while communicating through shared variables. Unlike SkyBlue, Ultraviolet is not a solver itself, but only coordinates the actions among its sub-solvers which include Blue (for functional LP), Indigo (for numeric inequalities), Purple (for simultaneous linear equalities), and Deep Purple (a partial solver for simultaneous linear equalities and inequalities; cf. QOCA and Cassowary in Section 3.4.2). One key advance of Ultraviolet was determining the order of invocation of sub-solvers to support constraint hierarchies—the outer loop for satisfaction is ordered by decreasing strength of constraints with each sub-solver potentially invoked multiple times [BFB98, p. 7].

Partitioning of the constraint graph is not only useful for increasing expressiveness but also for improving performance. The more sophisticated algorithms that support multi-output and cycles all have super-linear complexity, thus they may benefit from being subdivided into smaller

independent problems. Some evidence suggests that constraints in real applications tend to be modular, and therefore amenable to this kind of decomposition [VZV96].

3.2.4 Geometric Degrees of Freedom Analysis

Kramer's Geometric Constraint Engine (GCE) [Kra92] exploits symbolic analysis of geometric degrees of freedom which insulates the technique from the underlying representation and equations, and preserves the intuitive nature of the underlying problem. GCE's solver is given the task of constructing a "metaphorical assembly plan" (MAP) to describe how to satisfy a set of geometric constraints. Though Kramer presents his technique as novel (and it certainly seems superficially distinct from the other algorithms we have discussed), it is simply a local propagation algorithm at its essence. GCE proceeds by searching for free geometric entities, and selecting transformations to assign positions to those entities. It constructs the MAP in reverse order of ultimate execution, exactly as Sutherland's original LP algorithm for Sketchpad did. (In the forward direction, this can be seen as the propagation of rigidity; Brunkart calls this method contraction [Bru94]).

Kramer's propagation of geometric degrees of freedom is complicated by its need to infer the appropriate geometric transformation to fix (i.e., make rigid) a specific previously-free motion (in simple LP system, this requires only the evaluation of a pre-specified function, perhaps with some simple inference for multi-way numerical constraints). The planning for the MAP [BKH96], and the need to maintain a numerical model along with the symbolic geometric model distinguish GCE's geometric degrees of freedom analysis from other forms of local propagation.

3.2.5 LP strengths and weaknesses

Maximal efficiency and the ability to handle constraints over arbitrary domains are the primary strengths of local propagations algorithms. As previously mentioned, the key weakness of local propagation algorithms is their inability to simultaneously consider multiple constraints. These cycles must be managed by domain-specific techniques; more sophisticated local propagation solvers manage sub-solvers to provide this capability.

3.3 Iterative numeric solvers

Iterative numeric solvers have been used in constraint solving systems ever since Sketchpad. Their primary strength is that they are very general, and thus widely applicable. In particular, numeric techniques permit solving simultaneous non-linear constraints (such as maintaining equal lengths or distances) which arise often in geometric applications. Sutherland's Sketchpad exploits the representation of constraints directly in terms of the error, thus reducing constraint satisfaction to the well-studied problem of functional minimization. However, since iterative optimization techniques are often slow (their computational complexity is generally at least quadratic and the constant factors are relatively large), they are not particularly well-suited for interactive applications. Sutherland's relaxation technique is only used when his one-pass local propagation algorithm fails to find a solution [Sut63, p. 57]. ThingLab also relies on relaxation as a backup technique when faster methods fail [Bor79, p. 68–69].

Recognizing that constraint solving via iterative numeric techniques can be viewed as classical functional optimization opens up a world of techniques [Fle87]. Relaxation is simply an iterative hill climbing (or equivalently a gradient, or steepest descent) algorithm. These optimizers are reasonably good at finding a local minimum independent of the initial guess, but converge only linearly to the local minimum. More importantly, the technique only finds a *local* minimum, ignorant of the global search space.

Other systems' solvers, including Juno and Juno-2 [Nel85, HN94], use (multidimensional) Newton-Rhapson iteration to exploit derivative information. Some systems use automatic differentiation to relieve the user from specifying derivatives [GW93], and others simply limit the set of functions known to the underlying solver. Juno-2's solver performs numerous optimizations, including propagation of known state, unification of pair constraints, unpacking (to primitive constraints separating numeric constraints from non-numeric constraints) and re-packing (reducing the number of constraints and unknowns before passing them along to the Newton-Rhapson solver). Newton-Rhapson converges quadratically (faster than gradient descent), but relies on a sufficiently accurate initial guess and an invertible Jacobian.[14] The Levenberg-Marquardt method [BF85] dynamically weights a combination of Newton-Rhapson and gradient descent, permitting solvers to exploit the faster convergence of Newton-Rhapson once in the proximity of a local minimum; this hybrid solver is used in maintaining the constraints in the Chimera editor [KF92].

Besides being relatively inefficient, iterative numeric solvers pose other problems for interactive graphical application constraint solvers. Because of their iterative nature, it is sometimes difficult to tell if convergence is slow, or if the system is insatiable. Because the methods are local optimizers, the solution converged upon depends on the initial solution. Slight changes in the initial conditions can result in finding radically different solutions. This behaviour is almost never what the end-user expects.

Difficulty of implementation is yet another hindrance to the spread of iterative solving techniques. Coding iterative numeric constraint solvers is not for the numerically-challenged. Various numerical stability problems (e.g., singular or nearly-singular matrices) crop up repeatedly. Only with an arsenal of carefully combined sophisticated algorithms (e.g., singular value decomposition can be useful for under-constrained systems in place of Gaussian elimination) can the techniques perform computations robustly. Bramble and its "Snap-Together Mathematics" package provides some of these tools in the context of Whisper—an extensible Scheme-like language [Gle93, GW93].[15]

One of the more promising uses of iterative techniques is exemplified by the GLIDE interactive graph layout system [RMS97]. GLIDE gives up on the difficult (and inefficient) problem of global optimization of a graph layout. Instead, it focuses on exploiting the solver's strength—local minimization—and combining that with the interactive user's strength—global layout. To make this combination most useful, the numerical solver is physically-based, using a generalized spring model. The visual organization features (see Section 2.2) are mapped to sets of spring-like objects[16] among nodes. The energy minimization function uses varying spring-constants to provide preferential constraint satisfaction similar to constraint hierarchies or weighting of errors (as in QOCA [MCF98, BMSX97]).

GLIDE's iterative solver then simulates its physical model, trying to minimize the energy of the system. It uses Euler's method to compute the position and momentum of each node. During solving iterations the configuration is animated, and a kinetic energy threshold shuts down the system once it is stable, until the next user interaction. The animation reinforces the spring metaphor, and aids the user in establishing an accurate mental model. The collaborative approach of constraint solvers augmenting user interaction through physical models and understandable metaphors seems to counteract many of the difficulties iterative techniques otherwise experience. Differential methods are another physically-based technique; they are discussed in Section 3.5.

The constrained graph layout solver [HM96] takes a non-interactive approach and attempts to

[14] The Jacobian is the matrix of partial derivatives.

[15] SCWM uses a similar extensible language called Guile Scheme.

[16] They are not physically-precise springs (i.e., they can violate Hooke's Law) because some may have only a repulsive force.

perform global optimization of a spring-model energy function (a simplified aesthetic criterion) subject to arbitrary linear equality and inequality constraints. The first cost function He and Marriott consider, Model A, is a non-polynomial metric suggested by Kamada [HM96, p. 221]. Because this function is expensive to compute partial derivatives for (a significant cost in many iterative optimization algorithms) and lacks second-derivative continuity, He and Marriott propose Model B, a polynomial approximation to the first model. Their expectation is that the smoothness in the partial derivatives will permit better behaved solutions. The primary limitation of Model B was weakening of inter-node repulsive forces; this results in layouts where nodes overlap.

He and Marriot's layout algorithm is based on an active-set [Fle87] technique which is useful for optimizations constrained by inequalities. The active set method is also used by QOCA [BMSX97], and related to the simplex algorithm (see Section 3.4.1). As with simplex, finding an initial feasible solution for the active set method for graph layout requires additional work. Kamada's unconstrained algorithm simply puts the n nodes onto a regular n-polygon. He and Marriott augment this to simply find the least-squares closest solution which is feasible—this is a quadratic (and thus convex) programming problem, so any of the numerous applicable techniques suffices.[17] Their algorithm, while only of polynomial complexity, is slow on even small problems (a twenty node graph requires 33 seconds of computation on a 486DX/2-66).

Because of their generality, iterative numeric techniques are a useful method of last resort, and some of their uses for physical simulations seems promising. However, given the limited progress that has been made on general non-linear optimization techniques, it is likely that other, more restrictive, algorithms are a more useful direction to pursue in future work.

3.4 Direct numeric solvers

Direct numeric constraint solvers avoid the difficulties of iterative numeric solvers by attempting to find an exact solution through symbolic manipulation of the constraint equations. As with iterative numeric solvers, the domain for constraints is restricted to numbers. Additionally, to make solving manageable, direct numeric solvers further restrict the constraints they allow. The most common restriction is to permit only linear equality relationships—linear systems of equations have numerous applications, and there exist efficient algorithms for solving them.

The simplest algorithm for solving simultaneous linear systems of inequalities is Gaussian elimination. In the equations' matrix form, Gaussian elimination corresponds to computing the row-reduced form. From this triangular form a value for a variable can be read off a row directly, then that variable's value can be substituted into the other equations, and the process repeats. This back-substitution corresponds to the local-propagation solver's behaviour during the execution phase (its planning phase corresponds to choosing the ordering of rows for the back-substitution). The need to compute the row-reduced form arises from the desire to handle simultaneous systems (i.e., those involving cycles in the constraint graph). If there are no cycles, then Gaussian elimination is unnecessary and simple propagation of known-state (as LP solvers do) suffices.

Gaussian elimination only finds a unique solution when a system is fully specified (i.e., the corresponding matrix is of full rank) as with systems of n independent equalities with n variables.[18] In constraint systems, however, under-constrained systems are far more common.

[17] Tree layout as formulated by their Model C is also only a quadratic programming problem. Again, He and Marriott use a variant of the active set method.

[18] Independence assures that rows provide useful information; rows that are linear combinations of other rows are not helpful in constraining the system.

3.4.1 Simplex algorithm

As mentioned earlier, under-constrained systems require a means of disambiguating possible solutions. As we have seen, constraint hierarchies and optimization of a global error metric are two useful ways of declaratively specifying preferred solutions. This leads to inverting the problem: instead of talking about solving an under-constrained linear system, we can focus instead on the error function and describe our goal as optimizing an objective function subject to a set of constraints. Dantzig's famous simplex algorithm is a simple technique for optimizing a linear function subject to linear equality constraints [MS98, pp. 63–72]. Though simplex works only on equalities, an arbitrary inequality can be automatically rewritten using a non-negative slack variable. For example, $x > y$ becomes $x = y + s_1$, where the slack variable $s_1 \geq 0$—this last non-negativity restriction on s_1 applies to all variables in the simplex tableau (the matrix on which the algorithm operates).

The simplex algorithm is split into two phases. Phase I finds an initial solution to the constraints, and phase II finds an optimal solution. Consider the four constraints:

$$1 \leq x \quad \wedge \quad x \leq 3 \quad \wedge \quad 0 \leq y \quad \wedge \quad 2y - x \leq 3$$

These inequalities correspond to the darkened region of Figure 9. Since the optimization function is linear, the optimal score must occur at a vertex of the enclosing polygon. In terms of the picture, phase I finds any of those vertices (called a basic feasible solution), while phase II involves pivoting the system to move between adjacent vertices, systematically and efficiently searching for the optimal solution.

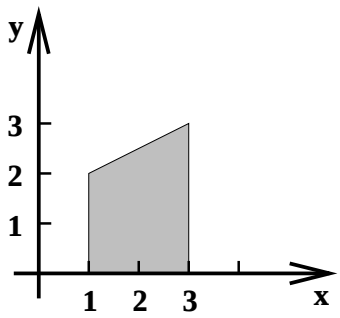

Figure 9: Simplex optimization problem [MS98, p. 64]

Finding an initial solution for simplex phase II is our original constraint satisfaction problem without the optimization criteria. In an interesting self-reference, we solve the constraints using the simplex algorithm on a modified problem. To avoid infinite regress, however, we must rely on a different technique for finding the initial basic feasible solution of this new problem. This is done by setting up our modified problem cleverly: given the initial constraints, we set each equation to zero by rearranging terms, and then replace the zeroes with a sequence of distinct artificial variables, and minimize the sum of those artificial variables. The artificial variables in this potential initial solution correspond to the errors in satisfying the original constraints. Most importantly, though, the modified system is already at a solution to the modified problem—that there is some (possibly zero) error in satisfying the original constraints. Phase II for this modified problem can proceed immediately in attempting to minimize the error, getting us closer to a

feasible solution to the original problem. If successful, all artificial variables are removed since they are zero;[19] if unsuccessful, the original constraint system is insatiable (i.e., over-constrained).

After phase II of the modified problem succeeds, we have a feasible, but not necessarily optimal, solution to the original constraint problem—we have completed phase I of the original problem. If we are only interested in any solution to a possibly under-constrained system, we need do no more; otherwise we can proceed with phase II of the original problem to optimize our objective relative to the original constraints, thus unambiguously achieving the solution we prefer (as declaratively specified by the objective function we chose).

3.4.2 QOCA and Cassowary: Incremental simplex

In Borning's spectrum of solvers, a variant of the simplex algorithm is dubbed "Orange," and an incremental version, DeltaOrange, is mentioned as a research direction [FBMB90]. Cassowary and QOCA are two variants of an incremental simplex algorithm [MCF98, BMSX97].

As one would imagine, Cassowary and QOCA are very similar to the batch simplex algorithm. Both lift the restriction of non-negativity on all variables by using two tableaus: an unrestricted tableau and a restricted, simplex tableau. Only the variables in the simplex tableau have the non-negativity restriction.[20] Cassowary and QOCA are incremental in that they permit adding and removing constraints while maintaining basic feasible solved form. Both algorithms proceed identically until the optimization (of the original problem) phase. Adding a constraint involves re-expressing inequalities as equalities, using an artificial variable to represent the error, and minimizing that error in the added equation. If the error cannot be minimized to zero, the new constraint is inconsistent and an exception is thrown. This is essentially an incremental version of simplex's phase I.

Removing a constraint is a bit more complicated because the effects of a single equation are spread throughout the tableaus as they are manipulated. This difficulty is overcome by creating a distinct "marker" variable for each constraint added to the tableau.[21] A marker variable indicates the effect of a constraint on the tableau, and that constraint can be removed by pivoting to make the marker variable basic, and then removing that row. Clearly, removing a constraint cannot make the system infeasible, so the tableau remains in basic feasible solved form.

The final incremental operation the algorithms provide is the ability to change a constraint. Often this is done for simple constraint equations which track, e.g., pointer movement. In Cassowary, these kinds of constraints are called "edit constraints." Usually changing a edit constraint's value requires only changing a constant in the tableau. Occasionally, the change will make the system infeasible; visually, this occurs when graphical objects first bump up against or leave other objects. This corresponds to a new configuration at an optimal but infeasible solution (i.e., it corresponds to an optimal point *outside* of the shaded region in Figure 9). When this occurs, the dual simplex algorithm is used to restore feasibility—to move from an infeasible and optimal solution to a feasible and still optimal solution. Typically this procedure requires only a single pivot to restore feasibility. The efficiency of this operation is essential for interactive graphical applications to maintain fluid animation while the user directly manipulates the system.

The primary difference between QOCA and Cassowary is in how they choose among possible solutions to the constraint hierarchy—how they perform phase II optimization. Cassowary permits

[19] If an artificial variable is still basic (i.e., appearing only once in the tableau, alone on one side of an equation) after optimization of the modified problem, we can make it parametric (i.e., move it out of the basis) by pivoting.

[20] Since the optimization phase requires this restriction to find adjacent vertices, that phase of the algorithm is restricted to only the simplex tableau.

[21] In implementations, other variables guaranteed to appear only in a single equation (e.g., slack variables) are overloaded to serve as marker variables.

an error in each non-required constraint equation. Since the error can be either positive or negative, we need two error variables associated with each equation: δ^+ and δ^-. Two variables are required because the simplex algorithm's non-negativity restriction on variables would otherwise prevent the representation of negative errors. The optimization function is then chosen to be a weighted sum of these error variables. The weighting is determined by the preferences of the constraints using a constraint hierarchy specification. To ensure we satisfy one strong constraint in preference to numerous weaker constraints, the objective function uses symbolic weights and lexicographical ordering. Generally, weak stay constraints are added to force each variable to remain where it is; these constraint values are then updated after each optimization of the system so that future optimizations will keep the variables' values the same unless they must be altered by some stronger constraint.

Instead of using preferences on constraints to control the optimization function, QOCA uses a global least-squares better comparator. QOCA's goal is to minimize the weighted sum of the squares of the error of each variable relative to its desired position. For this technique, each variable has a preferred location (analogous to the stay constraint for Cassowary) and a numerical weight of how strong the preference is. QOCA then must solve the quadratic programming problem of minimizing $\sum w_i \delta_i^2$, where w_i is the weight of the ith variable, and δ_i is the error from its desired location.

Convex quadratic programming is well-studied and two algorithms have been considered for use by QOCA: the active set method (currently used), and linear complementary pivoting. Both algorithms are related to the simplex technique.

The active set method [Fle87] is an iterative technique which maintains an active set of the equality constraints and the subset of the inequalities that are tight in the sense that their slack variables are parametric. At each step in the iteration, we either move as far towards an optimal solution as possible while maintaining feasibility relative to some new inequality that we add to the active set, or we move more toward optimality by removing a constraint from the active set. When the active set can no longer by modified, we are at an optimal, feasible solution [BMSX97].

Linear complementary pivoting is another approach to solving convex quadratic optimization problems. This technique works by first introducing dual slack variables and dual variables. Each of these new variables is complementary to an existing variable in the primal (original) problem; the dual slack variables to the primal parametric variables and the dual varaiables to the primal basic variables. Then we augment the tableau of the primal problem with equations relating the dual slack variables to the sum of partial derivatives of the objective function with respect to the parametric variables and the dot product of rows of the primal problem with dual variables. By maintaining the property that complementary variables may not both be positive while pivoting this combined problem repeatedly, we achieve a feasible and optimal solution to the primal problem. Because the partial derivative of the quadratic objective function is linear, we can use simplex as a solution technique (this is similar to Gleicher's differential method technique—see Section 3.5). Borning et al. provide an illustrative example [BMSX97].

QOCA gives up the ability to express arbitrary constraints at varying preferences. It instead guarantees a variable-weighted least-squares-better solution to the under-constrained problem. This comparator is especially useful in geometric applications since it tries to place objects as close as possible to where they are desired to be. The weighting function can be used to control which objects should be placed closest to their desired positions. Cassowary, on the other hand, places weights on the constraints, not on the variables. This is more general as it permits specifying preferences about arbitrary constraints, not just about stay constraints.[22] QOCA's

[22]The formulation of the quasi-linear error metric in the description of Cassowary as embedded in QOCA's solving framework does not permit this generalization. There, the authors associate δ^+ and δ^- with variables instead of

least-squares comparator also comes at the price of using additional numerical techniques (computing the derivative symbolically) and further implementation complexity. Performance for both QOCA and Cassowary is good, handling re-solves (i.e., edit constraint changes) of systems of around 600 constraints and 700 variables in under 30ms on average [MCF98].

3.5 Differential methods

Gleicher's Bramble drawing program permits quadratic constraints to be expressed and solved efficiently by using an approach he calls differential methods. The differential method technique is enabled by limiting the problem only to *maintenance* of constraints that *already* hold. All other systems discussed use a "specify-then-solve" methodology where the solver is responsible both for producing an initial solution and for maintaining that solution as the system is perturbed. Instead, Bramble requires that the user initially establish the desired relationship before adding the corresponding constraint to the solver—augmented snap-dragging is the mechanism that aids the user in establishing a desired relationship, and simultaneously permits adding the constraint to those that the system will maintain (see Section 2.1).[23]

Offloading the establishment of the initial configuration from the constraint solver simplifies the solver's task—instead of maintaining relationships regarding the absolute positions of objects, differential manipulation relates the *motion* of objects. Since the motion of an object is described by its derivative with respect to time, quadratic relationships in position are reduced to linear relationships in derivatives. Maintenance of linear constraints is a far easier job (see Section 3.4). The linear systems are solved to minimize the derivative of the configuration. The one added step in Briar is to solve an ordinary differential equation after solving for the unknown time derivative; Euler's method is one simple technique for computing an absolute position from the initial conditions and the derivative.[24]

Another key benefit of differential manipulation is that it permits choosing an underlying representation of an object's state independent of the user-interface controls for that object. For Gleicher's Through-the-Lens Camera Control (TLCC), he expresses the three-dimensional location and orientation of the camera via quaternions [Sho85] which are much better behaved numerically, but far less intuitive to the user, and thus unsuitable for exposing directly [GW92]. Gleicher has also applied differential manipulation techniques to character animation systems[GL96].

4 Backtracking algorithms for constraint satisfaction

Backtracking search is one of the simplest and most common global search strategies over finite domains. The generic constraint satisfaction problem (CSP) consists of constraints C over n variables $x_1, x_2, ..., x_n$, each with a finite (possibly distinct) domain of allowable value assignments. To solve the CSP, an assignment, $\bar{a}$, must be found which associates each variable with a value from its corresponding domain such that C (the set of constraints) is satisfied [MCF98]. Backtracking is a means of systematically searching the space of possible solutions for such satisfying assignments. Green, one of Borning's spectrum of solvers, uses the related generate-and-test methodology (combined with local propagation) for its finite-domain constraints [FBMB90, p. 57].

constraints.

[23] Adding an arbitrary constraint to the system that is *not* already satisfied may be confusing to the user, so there is some evidence that this approach has usability benefits. Additionally, by knowing that the constraint is already satisfied, the solver need not worry about over-constrained systems [GW94].

[24] Animus also uses differential equations for the specification of continuous motion of objects being animated [BD86, Dui88].

Backtracking algorithms have generally not been used for user interface applications because of their poor performance. Yet, despite their exponential complexity, backtracking need not be discarded out-of-hand as unsuitable for the strict real-time requirements of interactive systems. SkyBlue [San94b] uses backtracking, but is able to maintain sufficient performance in the average case by using walk-bounds as a domain-specific pruning technique (see Section 3.2.3).

Numerous variants of the basic (called chronological) backtracking algorithm have been suggested and empirically analyzed for performance. The rest of this section summarizes a landmark paper by Kondrak and van Beek that describes a principled approach to evaluating backtracking algorithms [KvB97]. The next section discusses issues in using a backtracking approach to support disjunctions in incremental constraint solvers.

Kondrak and van Beek describe several backtracking algorithms. They analyze them in terms of two abstract performance measures: 1) the set of visited nodes in the search tree of possible assignments (Figure 10); and 2) the number of consistency checks needed (Figure 11). Since backtracking and consistency checking dominate the execution time of implementations of the algorithms, these analytical metrics allow us to more finely compare expected performance of algorithms.

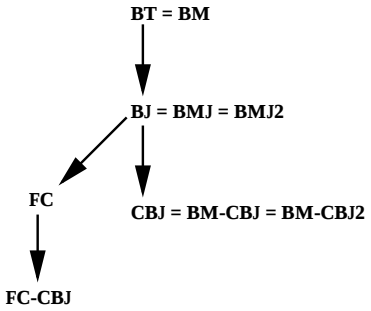

Figure 10: Kondrak and van Beek's hierarchy of the number of nodes visited for various backtracking algorithms. Edge $a \rightarrow b$ is in the graph if the set of nodes visited by algorithm b is always a subset of those visited by a [KvB97, p. 17, Figure 7].

Chronological backtracking (BT) is the simplest algorithm which assigns values to successive variables, checking for consistency only against already-assigned variables. The recursive algorithm backtracks when no instantiation for the current variable can preserve consistency—the backtracking returns to attempt the next domain value for the previous variable.

A simple improvement to BT is back-jumping (BJ); it prunes some of the search space by backtracking not to the immediately previous variable, but to the deepest past variable that has an assignment that conflicts against the current variable (re-assignments to the non-conflicting intermediate variables cannot jump us out from the dead-end). This pruning reduces the number of nodes visited. Conflict-directed back-jumping (CBJ) also back-jumps, but maintains a conflict set so that information gathered from further along in the tree is not discarded when back-jumping. CBJ can behave more cleverly than BJ in choosing to what node to backtrack, and thus may visit even fewer nodes. For all three of these algorithms, the consistency checks done at each node are the same (but the total number of consistency checks reduces as the algorithm improves from BT to BJ to CBJ). Both BJ and CBJ are backward-checking algorithms.

Forward checking (FC) filters the allowable domain of future variables based on the restrictions imposed by the current assignment. If any unassigned variable's domain is annihilated (i.e.,

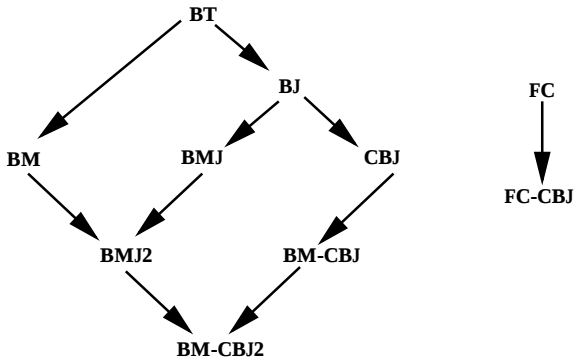

Figure 11: Kondrak and van Beek's hierarchy of the number of consistency checks for various backtracking algorithms. Edge $a \rightarrow b$ is in the graph if algorithm b never performs more consistency checks than a does [KvB97, p. 18, Figure 8].

there is no satisfying extension of the current assignment), FC backtracks chronologically.[25] FC's filtering lets it skip the same nodes that BJ avoids, but CBJ's conflict set can provide information permitting pruning that FC does not recognize. Thus FC visits a subset of the nodes that BJ visits, but not necessarily a subset of the nodes that CBJ visits. A combination of FC and CBJ, FC-CBJ, attempts to use information about variables that cause the current inconsistency to further prune the search space. FC-CBJ is shown to visit no more nodes (and do no more consistency checks) than FC, but could visit more nodes than CBJ (hence there is no edge CBJ $\rightarrow$ FC-CBJ in Figure 10. Kondrak and van Beek prove FC-CBJ correct (i.e., both sound in that it finds only solutions, and complete in that it finds all the solutions).

All of the above backtracking variants focus on reducing the number of nodes visited. While visiting fewer nodes can reduce the number of consistency checks required, some enhanced algorithms improve performance by reducing the number of consistency checks required. A back-marking scheme caches results of consistency checks to avoid the actual (often expensive) consistency check. Back-marking variants of BT, BJ, and CBJ exist and are called BM, BMJ, and BM-CBJ. Though each of these algorithms visits the same set of nodes as their corresponding non-back-marking cousin, each will perform no more consistency checks (and may require far fewer).

Though BMJ performs fewer consistency checks than BJ, it (somewhat surprisingly) may execute more checks than BM despite visiting fewer nodes. This results because the one-dimensional marking table cannot adequately maintain all the relevant information of the back-mark table. Intuitively, BM may have better caching behaviour than BMJ does. Kondrak and van Beek introduce the BMJ2 algorithm: an enhancement to BMJ that uses a two-dimensional marking table to ensure it never performs more consistency checks than BM does. BMJ2 still visits the same set of nodes as BJ and BMJ. BM-CBJ2, also introduced in the paper, is the analogous modification to BM-CBJ; it performs fewer consistency checks than both BMJ2 and BM-CBJ, while still visiting the same nodes as CBJ and BM-CBJ.

Though Kondrak and van Beek mostly limit their analysis to static ordering of variable instantiation, they do consider the popular minimum remaining values (MRV) heuristic which relaxes

[25] We can view known state propagation planning (see Section 3.2.1 and Section 3.2.2) as forward checking where annihilation of a domain corresponds to conflicting assignments (i.e., two edges pointing at the same node). While one-way LP solvers have no possible alternatives, multi-way solvers would need to backtrack if not for walk-strength annotations.

that restriction. All of the backtracking algorithms can be combined with the MRV, as suggested by Bacchus and van Run. The partial orders of Figures 10 and 11 both still hold, but can be strengthened using a result from Bacchus and van Run. They note that MRV makes backjumping redundant. This fact collapses the top five nodes (BT and BJ and their derivatives) in the node hierarchy, and merges BT+MRV/BJ+MRV and BM+MRV/BMJ+MRV/BMJ2+MRV in the consistency checks hierarchy [KvB97].

5 Incremental approaches for backtracking

None of the interactive graphical applications discussed in Section 2 permit disjunctive constraints. The ability to express the desire that either C_1 or C_2 (or both) holds is useful in geometric applications. For web page layout, we may want to ensure that two figures are horizontally adjacent and non-overlapping, but not care which is on the left and which on the right. Similarly, users of window managers may want a window to stay at either the top of the screen or the bottom of the screen [CSI86, p. 42]. To see how backtracking algorithms apply, let us formalize our problem as an extension to a simultaneous linear equality and inequality solver such as Cassowary [BB98].

We would like to be able to solve constraint systems of the form:

$$\begin{array}{llllllll} (C_{1,1} & \vee & C_{1,2} & \vee & \cdots & \vee & C_{1,d_1}) & \wedge \\ (C_{2,1} & \vee & C_{2,2} & \vee & \cdots & \vee & C_{1,d_2}) & \wedge \\ & & & \vdots & & & & \\ (C_{n,1} & \vee & C_{n,2} & \vee & \cdots & \vee & C_{n,d_n}) & \end{array}$$

where each $C_{i,j}$ is a linear equality or inequality constraint. Of course, such a system with $\forall i \in 1..n, d_i = 1$ is the conjunction-only incremental simplex algorithm discussed in Section 3.4.2; also, it is not necessary that each disjunction have the same number of terms.

If we ignore completeness, it is trivial to extend conjunctive solvers to soundly solve such a system: we require all the constraints to hold. Obviously, we would like to exploit the disjunctions to admit more solutions so that the solver is less restrictive (i.e., more complete). Since only one constraint per disjunctive set needs to be satisfied, we need only choose one constraint from each row to be satisfied. (Certainly more than one can be true, but we need only require one.) That is, the constraint system above is satisfied if and only if $\forall i \in 1..n,\ \exists s_i \in 1..d_i$ such that C_{i,s_i} holds. The s_i variables are selectors which choose which constraint of row i is satisfied; the domain of s_i is $1..d_i$. We write an assignment to a subset of variables, as a tuple: e.g., $\bar{a} = (s_1 \rightarrow 2, s_3 \rightarrow 1)$. In an arbitrary assignment, not all variables need be given values, but a solution to the system requires a full and consistent assignment.

For an assignment to be consistent, the set of selected linear equality and inequality constraints must be soluble. In the context of an incremental simplex solver such as Cassowary, this corresponds to successfully adding the selected constraints into the tableau. The incremental simplex algorithm can add and remove constraints efficiently, so chronological backtracking can be used to solve the system of conjunctions of disjunctions. We backtrack by removing a constraint, and descend down the search tree by adding a constraint as we make the corresponding new variable assignment.

Using the batch chronological backtracking algorithm is inefficient. We cannot expect real-time performance as we are dragging a point around the screen if a full backtracking solve must be performed for each iteration. We must make the algorithm incremental, and can learn from backtracking enhancements summarized in Section 4 and common incremental techniques described in

Section 3.

A simple version of the minimum remaining values optimization is obviously useful. A degenerate disjunction with a single constraint has a selector variable with only one element in its domain. The selected constraint must be satisfied in the tableau for the system to be consistent, so we can treat these constraints as we do in the conjunction-only situation. Thus, $\forall d_i = 1$ we should leave $C_{i,1}$ in the tableau permanently (until it is removed by the end-user). An alternate way (more natural in the implementation) to view this simplification is to partition the disjunction constraints and only introduce selector variables for those.

For another optimization, we must observe that Cassowary and QOCA's incremental simplex algorithm permits discovery of a conflict set of an added constraint. Recall that to add a constraint we introduce an artificial variable and equate it to the error in that constraint, then minimize the artificial variable (and hence the error) to zero using simplex optimization (see Section 3.4.2). If that process fails, the artificial objective will have a non-zero constant, and the marker variables appearing there correspond to a set of constraints that are inconsistent with the constraint we were trying to add. This is a conflict set, and can be used to support back-jumping (BJ) and CBJ. Though I'm not yet convinced that this conflict set is minimal (in any sense), it need not be to be correct—it may be a superset of the real conflict. To be a useful optimization when combined with BJ, the conflict set must be sufficiently small relative to the total number of constraints that it can permit useful pruning of the search tree.[26]

Another possibility for better supporting backtracking in interactive constraint solvers is preserving data structures from prior solutions. In particular, if a constraint has not changed, the unchanged constraints that conflicted with it will still conflict. A complication in exploiting this fact is that edit and stay constraints change frequently, potentially invalidating our prior knowledge.

Constraint hierarchies also complicate the use of disjunctions. Consider a non-required disjunction $C_{i,1} \vee C_{i,2}$. When we add $C_{i,1}$ to the tableau (because we are trying the assignment $s_i \rightarrow 1$), we need not make it a required constraint. Similarly, for the alternative assignment, $C_{i,2}$ need not be required. If neither can be satisfied exactly (i.e., an error variables of each, when added, is non-zero) which should be selected as active?

One answer also suggests a faster way of adding and removing effects of constraints (i.e., reducing the time to move through the search tree). The constraints in a disjunction could all be maintained in the tableau, and only their weights[27] manipulated. If $s_i = 2$, then the weight of $C_{i,2}$ would be set to 1.0, and $C_{i,j}, j \neq 2$ would each get zero weights. Dynamically changing constraint weights and re-optimizing seems like a plausible implementation strategy for controlling the effects of non-required disjunctions. Unlike adding and removing the constraint from the tableau, it does not require pivoting, and thus can execute more quickly.

Other possibilities to improve interactive backtracking performance include exploiting domain-specific knowledge and performing additional preprocessing of the stable constraint set. Other notions of consistency may be useful in guiding this research direction [DvB97, vBD97]. The latter technique could perhaps recognize that a certain pair of consistent configurations toggle in applicability as an object passes through a line or some other state changes. This corresponds to moving directly from a consistent leaf node to another consistent leaf node in the search tree. Extending the search tree with annotated transition edges would reduce the time spent backtracking and re-descending down the search tree. Alternatively, online statistics could be used to dynamically update transition weights or other prediction techniques could be used [ABG+94].

[26] When an experimental implementation of this technique was applied to an LPSAT engine that performs consistency checks using Cassowary, a factor of ten speedup was observed [WW98].

[27] Weights in Cassowary are coefficients of symbolic strengths.

Support for disjunctions can be added incrementally. An initial implementation is straightforward, but inefficient. As our experience using disjunctive constraints in geometric layout applications increases, we can select the optimizations to best handle the kinds of disjunctions we find most valuable.

6 Conclusion

Interactive graphical applications have explored using constraints for over thirty-five years, yet none are completely successful, and numerous challenges remain. Two important problems not yet well-addressed and not considered in-depth here include debugging constraints and reuse of solvers. Debugging constraint systems is challenging, and must be made easier for users [Gle95, San94a]. Constraint solving libraries must be developed so that the implementation effort for application programmers is minimized—software engineering research on system architectures [MT] and solvers that stress simple and efficient[28] interfaces [MCF98] can be exploited to improve this situation.

This paper surveys several interactive graphical application domains that use constraint systems. Table 1 shows that several kinds of constraints are especially relevant for geometric applications, but that the constraints provided by an application are highly influenced and restricted by its underlying solver. Thus, increasing expressiveness of constraint solvers is a primary concern. The fundamental challenge is to not sacrifice performance while expanding the class of constraints that solvers handle. Figure 1 relates all of the solvers by their techniques and expressiveness. Understanding the evolution of techniques, recognizing the similarities among the approaches, and considering novel combinations of various systems exposes many areas for future work.

Extensible solvers are a useful and flexible mechanism for exploiting domain-dependent optimizations while retaining generality. In particular, Ultraviolet [BFB98] provides a useful framework for embedding solvers, but does not have an integrated, fully general linear equality and inequality sub-solver. Combining Cassowary into the Ultraviolet system is a straightforward and likely useful extension.

The relationships in interactive systems are often dynamic. Applications including web page layout and window management can benefit from the automatic management of symbolic constraint relations. The work on paths [Bor79] and pointer variable extensions for constraint systems [VZMGS91, VZMGS] may provide the basis for a similar system for Cassowary or Ultraviolet.

Physically-based systems such as the spring-layout of GLIDE [RMS97] and differential methods of Briar [GW94, GW92] demonstrate advantages of using simulation-based solvers. Animating other solvers' solution processes may be beneficial to creating a seemingly more responsive system, and providing a more understandable solution due to the physical metaphor. The collaborative aspect of these two systems is also instructive: constraint solving technology need not do everything. Users are good at direct manipulation and at global search and interactive systems should permit leveraging those abilities

Finally, Section 5 discusses some ideas on how backtracking performance can be improved for more general interactive constraint solvers. In particular, the ideas presented there will likely grow into support for disjunctions in Cassowary, and possibly other solving systems.

Interactive graphical applications can benefit dramatically from fast, expressive, understandable, and reusable constraint solvers. Improving constraint satisfaction algorithms in these direc-

[28] The Janus In Motion (JIM) application communicates with its Parcon constraint solver via Unix named pipes. Though this design certainly decouples the constraint solver from the application, the performance cost is hard to accept in an interactive application [GLM+96].

tions is important if we are to fully exploit the benefits that the declarative nature of constraints can provide.

References

[ABG+94] Salmon Azhar, Greg Badros, Arman Glodjo, M. Kao, and John Reif. Data compression techniques for stock market prediction. In *Proceedings of 1994 Data Compression Conference*, pages 72–82, Snowbird, UT, March 1994.

[BB98] Greg Badros and Alan Borning. The cassowary linear arithmetic constraint solving algorithm: Interface and implementation. Technical Report UW-CSE-98-06-04, University of Washington, Seattle, Washington, June 1998.

[BD86] Alan Borning and Robert Duisberg. Constraint-based tools for building user interfaces. *ACM Transactions on Graphics*, 5(4):345–374, October 1986.

[BF85] Richard L. Burden and J. Douglas Faires. *Numerical Analysis*. PWS Publishing Company, Boston, Massachusetts, fifth edition, 1985.

[BFB98] Alan Borning and Bjorn Freeman-Benson. Ultraviolet: A constraint satisfaction algorithm for interactive graphics. *Constraints: An International Jounal*, 3:1–26, 1998.

[BKH96] Sanjay Bhansali, Glenn A. Kramer, and Tim J. Hoar. A principled approach toward symbolic geometric constraint satisfaction. *Journal of Artificial Intelligence Research*, 4:419–443, 1996.

[BLM97] Alan Borning, Richard Lin, and Kim Marriott. Constraints for the web. In *Proceedings of 1997 ACM Multimedia Conference*, 1997.

[BMMW89] Alan Borning, Michael Maher, Amy Martindale, and Molly Wilson. Constraint hierarchies and logic programming. In *Proceedings of the Sixth International Conference on Logic Programming*, pages 149–164, Lisbon, June 1989.

[BMSX97] Alan Borning, Kim Marriott, Peter Stuckey, and Yi Xiao. Solving linear arithmetic constraints for user interface applications: Algorithm details. Technical Report 97-07-01, University of Washington, Seattle, WA, September 1997.

[Bor79] Alan Borning. *ThingLab—A Constraint-Oriented Simulation Laboratory*. PhD thesis, Stanford University, March 1979. A revised version is published as Xerox Palo Alto Research Center Report SSL-79-3 (July 1979).

[Bro88] Arun N. Brotman, Lynne Shapiro ann Netravali. Motion interpolation by optimal control. In *Proceedings of SIGGRAPH 1988*, pages 309–315, Atlanta, GA, August 1988.

[Bru94] Mark W. Brunkhart. Interactive geometric constraint systems. Master's thesis, University of California, Berkeley, Berkeley, California, May 1994.

[BS86] Eric. A. Bier and Maureen C. Stone. Snap-dragging. In *Proceedings of SIGGRAPH 1986*, Dallas, August 1986.

[BS98] Greg Badros and Maciej Stachowiak. Scwm—the scheme constraints window manager. Web page, 1997–1998. http://huis-clos.mit.edu/scwm/.

[CM95] Sitt Sen Chok and Kim Marriott. Automatic construction of user interfaces from constraint multiset grammars. In *Proceedings of IEEE International Symposium on Visual Languages*, pages 242–249, Los Alamitos, CA, September 1995.

[CM98] Sitt Senn Chok and Kim Marriott. Automatic construction of intelligent diagram editors. In *Proceedings of UIST 1998*, San Francisco, CA, November 1998.

[Cru95] Isabel F. Cruz. Expressing constraints for data display specification: A visual approach. In *Principles and Practice of Constraint Programming*, chapter 23, pages 445–469. MIT Press, Cambridge, Massachusetts, 1995.

[CSI86] Ellis S. Cohen, Edward T. Smith, and Lee A. Iverson. Constraint-based tiled windows. *IEEE Computer Graphics and Applications*, pages 35–45, May 1986.

[DBETT94] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. Algorithms for drawing graphcs: An annotated bibliography. *Computational Geometry: Theory and Applications*, 4:235–282, 1994.

[DBETT99] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice-Hall, Englewood Cliffs, New Jersey, 1999.

[DFM93] Ed Dengler, Mark Friedell, and Joe Marks. Constraint-driven diagram layout. In *Proceedings of the 1993 IEEE Symposium on Visual Languges*, pages 330–335, Bergen, Norway, August 1993.

[Dui88] Robert Adámy Duisberg. Animation using temporal constraints: An overview of the Animus system. *Human-Computer Interaction*, 3:275–307, 1987-1988.

[DvB97] Rina Dechter and Peter van Beek. Local and global relational consistency. *Theoretical Computer Science*, 173:283–308, 1997.

[EL88] Danny Epstein and Wilf LaLonde. A Smalltalk window system based on constraints. In *Proceedings of the 1988 ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 83–94, San Diego, September 1988. ACM.

[FB93] Barry Fowler and Richard Bartels. Constraint-based curve manipulation. *IEEE Computer Graphics and Applications*, pages 43–49, September 1993.

[FBMB90] Bjorn N. Freeman-Benson, John Maloney, and Alan Borning. An incremental constraint solver. *Communications of the ACM*, 33(1):54–63, January 1990.

[FBWB92] Bjorn Freeman-Benson, Molly Wilson, and Alan Borning. DeltaStar: A general algorithm for incremental satisfaction of constraint hierarchies. In *Eleventh Annual International Phoenix Conference on Computers and Communications*, pages 561–568, Phoenix, AZ, April 1992.

[Fle87] Roger Fletcher. *Practical Methods of Optimization*. John Wiley and Sons, New York, 1987.

[GL96] Michael Gleicher and Peter Litwinowicz. Constraint-based motion adaptation. Technical Report TR 96-153, Apple Computer, June 1996.

[Gle92] Michael Gleicher. Integrating constraints and direct manipulation. In *Proceeding 1992 Symposium on Interactive 3D*, pages 171–174, 1992.

[Gle93] Michael Gleicher. A graphics toolkit based on differential constraints. In *Proceedings of UIST 1993*, pages 109–120, Atlanta, GA, November 1993.

[Gle95] Michael Gleicher. Practical issues in graphical constraints. In *Principles and Practice of Constraint Programming*, chapter 21, pages 407–426. MIT Press, Cambridge, Massachusetts, 1995.

[GLM+96] P. Griebel, G. Lehrenfeld, W. Mueller, C. Tahedl, and H. Uhr. Integrating a constraint solver into a real-time animation environment. Proceedings of IEEE Symposium on Visual Languages, September 1996.

[Gos83] James Gosling. *Algebraic Constraints*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, May 1983.

[GW92] Michael Gleicher and Andrew Witkin. Through-the-lens camera control. In *Proceedings of SIGGRAPH 1992*, July 1992.

[GW93] Michael Gleicher and Andrew Witkin. Supporting numerical computations in interactive contexts. In *Graphics Interface 1993*, 1993.

[GW94] Michael Gleicher and Andrew Witkin. Drawing with constraints. *Visual Computer*, 11(1):39–51, 1994.

[Hen92] Tyson R. Henry. *Interactive Graph Layout: The Exploration of Large Graphs*. PhD thesis, University of Arizona, Tucson, Arizona, June 1992. Also TR-92-03.

[HH91] Tyson R. Henry and Scott E. Hudson. Interactive graph layout. In *Proceedings of UIST 1991*, pages 55–64, November 1991.

[HHMV95] Richard Helm, Tien Huynh, Kim Marriott, and John Vlissides. *An Object-Oriented Architecture for Constraint-Based Graphical Editing*, chapter 14, pages 217–238. Springer, 1995.

[HM90] Scott E. Hudson and Shamim P. Mohamed. Interactive specification of flexible user interface displays. *ACM Transactions on Information Systems*, 8(3):269–288, July 1990.

[HM96] Weiqing He and Kim Marriott. Constrainted graph layout. In S. North, editor, *Proceedings of 1996 Graph Drawing Conference*, pages 217–232, Berkeley, CA, September 1996. Springer Verlag.

[HMT+94] Hiroshi Hosobe, Ken Miyashita, Shin Takahashi, Satoshi Matsuoka, and Akinori Yonezawa. Locally simultaneous constraint satisfaction. In Alan Borning, editor, *Principles and Practice of Constraint Programming 1994*, pages 51–62, Orcas Island, WA, 1994.

[HN94] Allan Heydon and Greg Nelson. The Juno-2 constraint-based drawing editor. Technical Report 131a, Digital Systems Research Center, Palo Alto, California, December 1994.

[HS93] Scott E. Hudson and John T. Stasko. Animation support in a user interface toolkit: Flexible robust and reusable abstractions. In *Proceedings of UIST 1993*, pages 57–67, Atlanta, GA, November 1993.

[IMKT97] Takeo Igarashi, Satoshi Matsuoka, Sachiko Kawachiya, and Hidehiko Tanaka. Interactive beautification: A technique for rapid geometric design. In *Proceedings of UIST 1997*, pages 105–114, Banff, Alberta, Canada, October 1997.

[JMSY92] Joxan Jaffar, Spiro Michaylov, Peter J. Stuckey, and Roland H. C. Yap. The CLP($\Re$) language and system. *ACM Transactions on Programming Languages and Systems*, 14(3):339–394, July 1992.

[KF91] David Kurlander and Steven Feiner. Inferring constraints from multiple snapshots. Technical Report CUCS-008-91, Columbia University, New York, NY, May 1991.

[KF92] David Kurlander and Steven Feiner. Interactive constraint-based search and replace. In *CHI 1992 Proceedings*, May 1992.

[KK91] Tomihisa Kamada and Satoru Kawai. A general framework for visualizing abstract objects and relations. *ACM Transactions on Graphics*, 10(1):1–39, January 1991.

[KMS94] Corey Kosak, Joe Marks, and Stuart Shieber. Automating the layout of network diagrams with specified visual organization. *IEEE Transactions on Systems, Man, and Cybernetics*, 24(3):440–454, March 1994.

[Kra92] Glenn A. Kramer. A geometric constraint engine. *Artificial Intelligence*, 58(1–3):327–360, December 1992.

[KvB97] Grzegorz Kondrak and Peter van Beek. A theoretical evaluation of selected backtracking algorithms. *Artificial Intelligence*, 89(2):365–387, 1997.

[Mac91] Blair MacIntyre. A constraint-based approach to dynamic colour management for windowing interfaces. Master's thesis, University of Waterloo, Waterloo, Ontario, Canada, 1991.

[Mar94] Kim Marriott. Constraint multiset grammars. In *Proceedings of IEEE Symposium on Visual Language*, pages 118–125, Los Alamitos, CA, October 1994.

[MCF98] Kim Marriott, Sitt Sen Chok, and Alan Finlay. A tableau based constraint solving toolkit for interactive graphical applications. In *International Conference on Principles and Practice of Constraint Programming*, 1998.

[MGD+90a] Brad A. Myers, Dario Giuse, Roger B. Dannenberg, Brad Vander Zanden, David S. Kosbie, Philippe Marchal, Ed Pervin, Andrew Mickish, and John A. Kolojejchick. The Garnet toolkit reference manuals: Support for highly-interactive graphical user interfaces in Lisp. Technical Report CMU-CS-90-117, Computer Science Dept, Carnegie Mellon University, March 1990.

[MGD+90b] Brad A. Myers, Dario A. Giuse, Roger B. Dannenberg, Brad Vander Zanden, David S. Kosbie, Edward Pervin, Andrew Mickish, and Philippe Marchal. Garnet: Comprehensive support for graphical highly interactive user interfaces. *IEEE Computer*, November 1990.

[Mic98] Brian Michalowski. A constraint-based specification for box layout in CSS2. Technical Report UW-CSE-98-06-03, University of Washington, June 1998.

[MM95] Rich McDaniel and Brad A. Myers. Amulet's dynamic and flexible prototype-instance object and constraint system in C++. Technical Report CMU-CS-95-176, Carnegie Mellon University, Pittsburgh, PA, July 1995.

[MMM+97] Brad A. Myers, Richard G. McDaniel, Robert C. Miller, Alan S. Ferrency, Andrew Faulring, Bruce D. Kyle, Andrew Mickish, Alex Klimovitski, and Patrick Doane. The Amulet environment: New models for effective user interface software development. *IEEE Transactions on Software Engineering*, 23(6):347–365, June 1997.

[MMMF96] Brad Myers, Robert Miller, Rich McDaniel, and Alan Ferrency. Easily adding animations to interfaces using constraints. In *Proceedings of UIST 1996*, pages 119–128, Seattle, WA, November 1996.

[MS98] Kim Marriott and Peter J. Stuckey. *Programming with Constraints: An Introduction*. MIT Press, Cambridge, Massachusetts, 1998.

[MT] Nena Medvidovic and Richard N. Taylor. Reuse of off-the-shelf constraint solvers in C2-style architectures. Available from neno@ics.uci.edu.

[Nel85] Greg Nelson. Juno, a constraint-based graphics system. In *Proceedings of SIGGRAPH 1985*, San Francisco, July 1985.

[Not98] Michael Noth. Constraint drawing applet. Web page, 1998. http://www.cs.washington.edu/research/constraints/cda/info.html.

[OK98] Gregory M. Oster and Anthony J. Kusalik. ICOLA—incremental constraint-based graphics for visualization. *Constraints: An International Jounal*, 3:32–59, 1998.

[PVW85] T. Pavlidis and Christopher J. Van Wyk. An automatic beautifier for drawings and illustrations. In *Proceedings of SIGGRAPH 1985*, July 1985.

[RMS97] Kathy Ryall, Joe Marks, and Stuart Shieber. An interactive constraint-based system for drawing graphs. In *Proceedings of UIST 1997*, Banff, Alberta Canada, October 1997.

[San94a] Michael Sannella. Analyzing and debugging hierarchies of multi-way local propagation constraints. In Alan Borning, editor, *Principles and Practice of Constraint Programming 1994*, pages 63–77, Orcas Island, WA, 1994.

[San94b] Michael Sannella. The SkyBlue constraint solver and its applications. In *Proceedings of the 1994 Workshop on Principles and Practice of Constraint Programming*, Cambridge, Massachusetts, 1994. MIT Press.

[Sch83] Ben Schneiderman. Direct manipulation: A step beyond programming languages. *IEEE Computer*, 16(8):57–69, August 1983.

[Sho85] Ken Shoemake. Animating rotation with quaternion curves. In *Proceedings of SIGGRAPH 1985*, pages 245–254, San Francisco, CA, July 1985.

[SMFBB93] Michael Sannella, John Maloney, Bjorne 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.

[Sut63] Ivan Sutherland. *Sketchpad: A Man-Machine Graphical Communication System*. PhD thesis, Department of Electrical Engineering, MIT, January 1963.

[TMM+98] Shin Takahashi, Satoshi Matsuoka, Ken Miyashita, Hiroshi Hosobe, and Tomihisa Kamada. A constraint based approach for visualization and animation. *Constraints: An International Jounal*, 3:61–86, 1998.

[vBD97] Peter van Beek and Rina Dechter. Constraint tightness and looseness versus local and global consistency. *Journal of the ACM*, 1997.

[VW82] Christopher J. Van Wyk. A high-level language for specifying pictures. *ACM Transactions on Graphics*, 1(2):163–182, April 1982.

[VZ96] 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.

[VZMGS] Brad Vander Zanden, Brad A. Myers, Dario A. Giuse, and Pedro Szekely. Integrating pointer variables into one-way constraint models.

[VZMGS91] Brad Vander Zanden, Brad A. Myers, Dario A. Giuse, and Pedro Szekely. The importance of pointer variables in constraint models. In *Proceedings of UIST 1991*, pages 155–164, Hilton Head, SC, November 1991.

[VZV96] Brad Vander Zanden and Scott A. Venckus. An empirical study of constraint usage in graphical applications. In *Proceedings of UIST 1996*, pages 137–146, Seattle, WA, November 1996.

[WK88] Andrew Witkin and Michael Kass. Spacetime constraints. In *Proceedings of SIGGRAPH 1988*, pages 159–168, Atlanta, GA, August 1988.

[WW98] Steve Wolfman and Dan Weld. The LPSAT engine and its application to resource planning. To be submitted, IJCAI'99., 1998.