A Modular Geometric Constraint Solver for User Interface Applications

Hiroshi Hosobe
National Institute of Informatics
2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo 101-8430, Japan
hosobe@nii.ac.jp

ABSTRACT

Constraints have been playing an important role in the user interface field since its infancy. A prime use of constraints in this field is to automatically maintain geometric layouts of graphical objects. To facilitate the construction of constraint-based user interface applications, researchers have proposed various constraint satisfaction methods and constraint solvers. Most previous research has focused on either local propagation or linear constraints, excluding more general nonlinear ones. However, nonlinear geometric constraints are practically useful to various user interfaces, e.g., drawing editors and information visualization systems. In this paper, we propose a novel constraint solver called Chorus, which realizes various powerful nonlinear geometric constraints such as Euclidean geometric, non-overlapping, and graph layout constraints. A key feature of Chorus is its module mechanism that allows users to define new kinds of geometric constraints. Also, Chorus supports "soft" constraints with hierarchical strengths or preferences (i.e., constraint hierarchies). We describe its framework, algorithm, implementation, and experimental results.

KEYWORDS: geometric constraints, soft constraints, constraint solvers, module mechanisms, graph layouts

INTRODUCTION

Constraints have been playing an important role in the user interface field since its infancy [27]. A constraint states a relationship to be maintained, and is usually expressed as a mathematical relation among variables. A prime use of constraints in this field is to obtain geometric layouts of graphical objects. Once the geometric relationship of objects is defined with constraints, a constraint solver will automatically maintain the relationship afterward, which will then result in setting necessary object properties such as positions and directions. This mechanism is effective particularly when the layout is difficult to program with a simple loop or recursion.

UIST'01 Orlando FLA

To facilitate the construction of constraint-based user interface applications, researchers have proposed various constraint satisfaction methods and constraint solvers. Particularly, they had studied local propagation (or dataflow) constraint solvers that perform computations according to input/output relationships among variables [5, 25]. Such local propagation solvers have an advantage that they efficiently handle many kinds of constraints. However, they impose a serious limitation that they can hardly deal with simultaneous constraints and inequalities. Therefore, researchers have recently proposed numerical solvers for linear equality and inequality constraints [2, 13, 19].

Although there has been much research on constraint solvers for user interfaces, even recent linear solvers are not sufficient for various real-world applications. In fact, linear solvers do not support *nonlinear* constraints such as $x \times y = z$, which earlier local propagation solvers could treat. Also, linear solvers do not handle Euclidean geometric constraints such as parallelism, perpendicularity, and distance equality. We can consider still other useful geometric constraints that local propagation and linear solvers can hardly handle. For example, non-overlapping constraints on two-dimensional rectangular boxes are difficult to treat, because they are "conditional" relations that cannot be defined with a fixed conjunctive set of inequality constraints. As another example, it is hard for these solvers to process constraints for general *graph layout*, since such a constraint cannot be expressed as a simple arithmetic relation.

Nonlinear geometric constraints are practically useful to various user interfaces. A major category is drawing editors [14, 21]. For instance, a drawing editor in an educational digital whiteboard system will require Euclidean geometric constraints to handle diagrams that can be drawn with rulers and compasses. Also, information visualization systems for information retrieval, databases, and programming may need to express information structures with general graph layouts [16, 20]. Nonlinear geometric constraints are effective in expressing dynamic properties of objects in three-dimensional spaces for virtual reality and video games [8]. Furthermore, we can fully expect that demands for such technologies will increase along with the speedup of computers and the enhancement of computer graphics.

In this paper, we propose a novel constraint solver called Chorus,[1] which realizes various powerful nonlinear geometric constraints. In developing Chorus, we set the following two major goals:

1. Enabling the easy introduction of new kinds of geometric constraints. Unlike linear constraints, nonlinear geometric constraints come in a variety of forms. Therefore, it is necessary to allow users to easily define their own kinds of geometric constraints.
2. Supporting "soft" constraints with hierarchical *strengths* or preferences (i.e., *constraint hierarchies* [5]). Soft constraints are useful for describing the default behavior of user interface applications; for example, a soft constraint for dragging an object layout should be ignored if the layout is about to move outside its permitted area such as a window. Many previous systems supporting local propagation and linear constraints have focused on realizing strengths. It is natural that we also need strengths even for nonlinear geometric constraints.

To achieve the first goal, Chorus provides a *module mechanism* that allows users to define new kinds of geometric constraints. We can add a new kind of arithmetic constraints (e.g., Euclidean geometric and non-overlapping constraints) by constructing a new constraint class with a method that evaluates how well given variable values satisfy constraints. Also, we can introduce a new kind of non-arithmetic (or pseudo) constraints (for, e.g., general graph layout) by developing an evaluation module that measures how well variable values conform to given constraint sets in yielding layouts.

To reach the second goal, Chorus adopts numerical nonlinear *optimization*, by which it minimizes violations of constraints according to their strengths. Although some kinds of geometric constraints suffer from local optimal but global non-optimal solutions, Chorus alleviates this problem by incorporating a *genetic algorithm* that stochastically searches for global optimal solutions. Also, it implements such numerical techniques as replaceable optimization modules.

This paper is organized as follows: We first present the overview of the Chorus constraint solver. Second, we describe its basic mathematical framework and how to model constraints. Next, we present its constraint satisfaction algorithm and module mechanism. We then provide its implementation, and show experimental results on its performance. After giving related work and discussion, we mention the conclusions and future work of this research.

THE CHORUS CONSTRAINT SOLVER

The Chorus constraint solver was designed for user interface construction. By default, it provides linear equality, linear inequality, edit (update a variable value), and stay (fix a variable value) constraints. It can be extended to support additional geometric constraints. Currently, it provides Euclidean geometric constraints such as parallelism, perpendicularity, and distance equality, non-overlapping constraints on rectangular boxes, and graph layout constraints based on the spring model [15].

To find appropriate solutions even in over-constrained situations, Chorus supports constraint strengths in a similar way to constraint hierarchies [5]; it obtains approximate solutions of constraint hierarchies solved with the criterion least-squares-better [2, 19]. Chorus externally provides four strengths `required`, `strong`, `medium`, and `weak`, and also internally processes two strengths `very strong` and `very weak`. It restricts the `medium` and `weak` strengths to basically linear constraints.

Chorus adopts an editing model that incrementally constructs constraint systems by adding or removing constraints; that is, when a user wants to process graphical objects, Chorus allows the user to obtain a solution by adding/removing necessary constraints to/from the corresponding constraint system. Also, Chorus enables the user to repeatedly update variable values via edit constraints, which typically facilitates operations for moving objects.

The application programming interface for writing such operations was designed to provide a certain compatibility with a recent linear solver called Cassowary [2]; it allows a user to process a constraint system by creating variables and constraints as objects, and adding/removing constraint objects to/from the constraint solver object.

BASIC FRAMEWORK

This section provides our basic framework for modeling constraints and constraint systems. In this framework, it uses $\boldsymbol{x}$ to represent a variable vector $(x_1, x_2, \ldots, x_n)$ of n variables, and also $\boldsymbol{v}$ to indicate a variable value vector $(v_1, v_2, \ldots, v_n)$ of n real numbers (v_i expresses the value of x_i).

Constraints

To support various geometric constraints in a uniform manner, Chorus adopts *error functions* as a means of expressing constraints. An error function $e(\boldsymbol{x})$ is typically associated with a single arithmetic constraint (sometimes with a set of non-arithmetic constraints), and is defined as a function from variable value vectors to errors expressed as non-negative real numbers; that is, $e(\boldsymbol{v})$ gives the error of the associated constraint(s) for $\boldsymbol{v}$. For an ordinary error function associated with an arithmetic constraint, it returns a zero if and only if the constraint is exactly satisfied.

We assume that, for each $e(\boldsymbol{x})$, its gradient is known:

$$\nabla e(\boldsymbol{x}) = \left(\frac{\partial e}{\partial x_1}, \frac{\partial e}{\partial x_2}, \ldots, \frac{\partial e}{\partial x_n} \right)$$

However, whether gradients are actually used depends on the numerical optimization techniques. For example, a quasi-Newton method requires gradients, while Powell's method

[1] Chorus stands for "Constraint hierarchy optimization and resolution system."

does not [4]. Usually, the use of gradients will result in faster computation.

Constraint Systems

In the same way as constraint hierarchies [5], constraint systems in our framework can be divided into *levels* consisting of constraints with equal strengths. Constraints with the strongest preference are said to be *required* (or hard), and are guaranteed to be always satisfied (if it is impossible, there will be no solution). By contrast, constraints with weaker preferences are said to be *preferential* (or soft), and may be relaxed if they conflict with stronger constraints.

Solutions of constraint systems are defined as follows: let $e_{i,j}(\boldsymbol{x})$ be the error function of the j-th constraint ($1 \leq j \leq m_i$) at strength level i ($0 \leq i \leq l$); then solutions $\boldsymbol{v}$ are determined with the following optimization problem:

$$\begin{array}{ll} \underset{\boldsymbol{v}}{\text{minimize}} & E(\boldsymbol{v}) \\ \text{subject to} & e_{0,j}(\boldsymbol{v}) = 0 \quad (1 \leq j \leq m_0) \end{array} \tag{1}$$

where $E(\boldsymbol{x})$ is an objective function defined as

$$E(\boldsymbol{x}) = \sum_{i=1}^{l} \sum_{j=1}^{m_i} w_i e_{i,j}(\boldsymbol{x})$$

in which w_i indicates the weight associated with strength i, and the relation $w_1 \gg w_2 \gg \cdots \gg w_l$ holds. In this formulation, level 0 corresponds to required constraints, and the others to preferential ones. Intuitively, more weighted (or stronger) preferential constraints should be more satisfied.

Our framework simulates constraint hierarchies. Particularly, if the squares of constraint violations are used to compute error functions, a system in our framework will obtain approximate solutions of the similar hierarchy solved with the criterion least-squares-better [2, 19]. The largest difference is that a system in our framework slightly considers a weak constraint inconsistent with a stronger satisfiable one in computing its solutions, while the similar hierarchy would discard such a weak one. However, the approximation will become better if each weight w_i is much larger than w_{i+1}.

MODELING CONSTRAINTS

This section describes how to actually define constraints in our framework.

Arithmetic Equality Constraints

We can naturally model ordinary arithmetic equality constraints by using error functions. Usually, it is a simple task that consists of computing the squares of the difference between the left- and right-hand sides of the original equations.

For example, the Euclidean geometric constraint that forces the distance between points (x_i, x_j) and $(x_{i'}, x_{j'})$ to be x_k can be built as

$$e(\boldsymbol{x}) = \left(\sqrt{(x_i - x_{i'})^2 + (x_j - x_{j'})^2} - x_k \right)^2. \tag{2}$$

As another example, a constraint on an angle between two line segments is defined as follows: Let $\boldsymbol{p}_1 = (x_{i_1}, x_{j_1})$, $\boldsymbol{p}_2 = (x_{i_2}, x_{j_2})$, $\boldsymbol{p}_3 = (x_{i_3}, x_{j_3})$, and $\boldsymbol{p}_4 = (x_{i_4}, x_{j_4})$. Suppose that the constraint equates the angle between vectors $(\boldsymbol{p}_2 - \boldsymbol{p}_1)$ and $(\boldsymbol{p}_4 - \boldsymbol{p}_3)$ to x_k. Then we can define its error function as

$$e(\boldsymbol{x}) = \left(\frac{(\boldsymbol{p}_2 - \boldsymbol{p}_1) \cdot (\boldsymbol{p}_4 - \boldsymbol{p}_3)}{|\boldsymbol{p}_2 - \boldsymbol{p}_1||\boldsymbol{p}_4 - \boldsymbol{p}_3|} - \cos x_k \right)^2 \tag{3}$$

where "$\cdot$" indicates the dot (inner) product of two vectors.

Conditional Constraints

In the previous subsection, we could express each error function by using a single expression. However, it is not always possible because some constraints require different expressions depending on conditions. We refer to such constraints as conditional constraints.

The simplest conditional constraints are inequalities. An error function for an inequality constraint can be defined by using two expressions associated with two cases: if the inequality holds for a given variable value, the error function returns a zero; otherwise, it returns the value of the constraint error in the same way as equality constraints.

A more complicated example is non-overlapping constraints on rectangular boxes (here we assume that the sides of each box are always either horizontal or vertical). Such non-overlapping constraints are quite useful since this kind of boxes are frequently used in current two-dimensional graphical user interfaces.

Consider a box with opposite vertices $\boldsymbol{p}_1 = (x_{i_1}, x_{j_1})$ and $\boldsymbol{p}_2 = (x_{i_2}, x_{j_2})$ ($\boldsymbol{p}_1 \leq \boldsymbol{p}_2$) and another with $\boldsymbol{p}_3 = (x_{i_3}, x_{j_3})$ and $\boldsymbol{p}_4 = (x_{i_4}, x_{j_4})$ ($\boldsymbol{p}_3 \leq \boldsymbol{p}_4$). Then the non-overlapping constraint on these two boxes can be expressed as

$$x_{i_2} \leq x_{i_3} \vee x_{i_4} \leq x_{i_1} \vee x_{j_2} \leq x_{j_3} \vee x_{j_4} \leq x_{j_1}. \tag{4}$$

To model non-overlapping constraints in our framework, we need to define an error function for (4). A possible definition of it is

$$e(\boldsymbol{x}) = \begin{cases} 0 & \text{if (4) holds} \\ (d_x d_y)^2 & \text{otherwise} \end{cases}$$

where

$$d_x = \min\{x_{i_2} - x_{i_3}, x_{i_4} - x_{i_1}\}$$
$$d_y = \min\{x_{j_2} - x_{j_3}, x_{j_4} - x_{j_1}\}.$$

Intuitively, d_x (d_y) indicates the smaller of the violations of $x_{i_2} \leq x_{i_3}$ and $x_{i_4} \leq x_{i_1}$ ($x_{j_2} \leq x_{j_3}$ and $x_{j_4} \leq x_{j_1}$), and decreasing the error function reduces d_x and/or d_y.

Non-Arithmetic Constraints

We sometimes need non-arithmetic constraints whose error functions cannot be represented as arithmetic expressions. A

typical example is node connection constraints for general graph layout that have been incorporated in certain user interface systems [16]. Since laying out general graphs needs to consider overall node positions, it is hardly possible to evaluate node connection constraints individually. Therefore, we can regard node connection constraints as being non-arithmetic.

To handle general graph layouts, we adopt force-directed algorithms (see [28] for a survey). Generally, they consider a dynamic system in which every two graph nodes are connected with a spring, and they obtain a set of node positions that minimizes the total energy of all the springs.

We use such energy functions as error functions in our framework; that is, instead of individually minimizing an error function for each constraint, we minimize a single "aggregate" error function for all graph layout constraints.

Now we present how to actually define an error function for graph layout constraints by using the spring model [15]: Let $\boldsymbol{p}_s = (x_{i_s}, x_{j_s})$ be the position of node s ($1 \leq s \leq S$). Then the error function is defined as

$$e(\boldsymbol{x}) = \sum_{s<t} \frac{1}{2} k_{st} (|\boldsymbol{p}_s - \boldsymbol{p}_t| - l_{st})^2$$

where $k_{st} = K/d_{st}^2$ and $l_{st} = Ld_{st}$ with a certain constant K, the ideal length L of a single edge, and the graph-theoretic shortest path distances d_{st} between nodes s and t (note that the model considers a spring between every pair of nodes connected indirectly as well as directly).

Remarks

In practice, we are restricted in defining error functions. For example, some may want to use the absolute values of constraint violations instead of their squares (e.g., $e(\boldsymbol{x}) = |\sqrt{(x_i - x_{i'})^2 + (x_j - x_{j'})^2} - x_k|$ instead of (2)). Although it is theoretically permissible, it is practically inappropriate because it results in a worse convergence at solutions. According to our experience, to use the squares of constraint violations is usually a good idea.

In modeling constraints, it should be better to guarantee the "smoothness" of their error functions. It can be verified by examining the continuity of their gradients. In particular, this is important for conditional constraints.

It is sometimes necessary to "scale" errors of constraints. We mainly need this to balance the effects of constraints with equal strengths. For example, our current implementation scales up the errors (3) of angle constraints by 100 times to match them with the errors (2) of distance constraints, which we determined empirically.

ALGORITHM

This section proposes an algorithm for solving constraint systems described above.

Processing Required Constraints

This algorithm first processes required linear equality constraints. It converts the optimization problem (1) into the following problem with a new objective function $\mathcal{E}(\boldsymbol{x}')$ by solving the required linear equality constraints and then eliminating possible variables:

$$\begin{array}{ll} \underset{\boldsymbol{y}'}{\text{minimize}} & \mathcal{E}(\boldsymbol{y}') \\ \text{subject to} & \epsilon_{0,j_k}(\boldsymbol{y}') = 0 \quad (1 \leq k \leq m_0') \end{array} \tag{5}$$

where m_0' is the number of the required constraints other than the linear equations, and each $\epsilon_{0,j_k}(\boldsymbol{x}')$ is an error function of such a required constraint.

We perform the conversion as follows: Without loss of generality, we can assume $\boldsymbol{x}' = (x_1, x_2, \ldots, x_{n'})$, which means that variables $x_{n'+1}, \ldots, x_n$ are eliminated from $\boldsymbol{x}$. First, the required linear equality constraints are transformed into

$$x_{n'+i} = f_i(\boldsymbol{x}') \quad (1 \leq i \leq n - n')$$

where each $f_i(\boldsymbol{x}')$ is a linear function. Then $\mathcal{E}(\boldsymbol{x}')$ is computed as

$$\mathcal{E}(\boldsymbol{x}') = E(x_1, \ldots, x_{n'}, f_1(\boldsymbol{x}'), \ldots, f_{n-n'}(\boldsymbol{x}')).$$

Also, each j-th element of the gradient $\nabla\mathcal{E}(\boldsymbol{x}')$ is

$$\frac{\partial \mathcal{E}}{\partial x_j} = \frac{\partial E}{\partial x_j} + \sum_{i=1}^{n-n'} \frac{\partial E}{\partial x_{n'+i}} \frac{\partial f_i}{\partial x_j}$$

where $\partial f_i / \partial x_j$ is the coefficient of x_j in $f_i(\boldsymbol{x}')$.

Local Search

To solve the optimization problem (5), this algorithm exploits numerical nonlinear optimization. The aim is to search for local optimal solutions since ordinary numerical optimization techniques cannot always find global optimal solutions.

An example of numerical optimization techniques is a quasi-Newton method [4], which is also known as a variable metric method. This technique is a fast iterative method that exhibits superlinear convergence. Since it excludes fruitless searches by utilizing its history, it is usually faster than straightforward Newton's method. The current Chorus constraint solver actually provides it as a numerical optimization technique.

The techniques for numerical optimization also need to handle required constraints other than linear equations, which cannot be eliminated with the preprocess for required constraints. However, since many techniques including the quasi-Newton method cannot realize this directly, such techniques treat the remaining required constraints as preferential ones with the special strength `very strong` instead.

The weight w_i associated with each preferential constraint must be determined according to the precision of the numerical optimization technique. For example, the number of the significant decimal figures of the quasi-Newton method

is at most nine when the 64-bit double precision is used. Therefore, Chorus assigns weights 32^4, 32^3, 32^2, 32^1, and 1 to strengths `very strong`, `strong`, `medium`, `weak`, and `very weak` respectively.[2] To know how much these weights affect solutions, suppose a system of `strong` constraint $x = 0$ and `medium` one $x = 100$. Then the unique solution will be obtained as $x = 3.0303\cdots(= 100/33)$ by minimizing $32^3(x-0)^2 + 32^2(x-100)^2$ $(= 32^2\{33(x - 100/33)^2 + 320000/33\})$. Thus the difference of strengths is obvious. According to our actual experience, this precision allows us to discriminate constraint strengths in most user interface applications.

Global Search

To alleviate the weakness of numerical optimization techniques suffering from local optimal solutions, we adopt a genetic algorithm [11, 17] that searches for global optimal solutions. Generally, a genetic algorithm is a stochastic search method that repeatedly transforms a population of potential solutions into another next-generation population.

Figure 1 shows a pseudo program of the genetic algorithm used here. The algorithm first generates the initial population of n potential solutions randomly. Then it locally optimizes them by using numerical optimization techniques such as the quasi-Newton method. After that, it repeatedly generates populations of n solutions by performing genetic operators called evaluation, selection, crossover, and mutation together with numerical optimization until it obtains a population whose solutions have fully converged.

```
genetic_algorithm() {
  Generate the initial population of n potential solutions;
  Apply numerical optimization to each potential solution;
  Evaluate each potential solution;
  while (the evaluation results have not converged) {
    Select n pairs of potential solutions;
    Crossover each pair to generate a new potential solution;
    Mutate each new potential solution with a certain probability;
    Apply numerical optimization to each new potential solution;
    Evaluate each new potential solution; }
  return the optimal solution; }
```

Figure 1: The pseudo program of the genetic algorithm for global searches.

We designed the genetic operators as follows:

Evaluation: The fitness of a potential solution $\boldsymbol{v}$ is evaluated with $\mathcal{E}(\boldsymbol{v})$.

Selection: n pairs of potential solutions called "parents" (used to generate new solutions) are selected from the current population with tournament selection [17]; to determine one parent, it first randomly picks up two potential solutions from the current population, and then chooses the better evaluated one as the parent (overlapping selections are permitted).

Crossover: A new potential solution called "offspring" is reproduced from a pair of parents $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$ by using linear crossover [11]; as an offspring, it randomly selects one from $(\boldsymbol{v}_1 + \boldsymbol{v}_2)/2$, $(-\boldsymbol{v}_1 + 3\boldsymbol{v}_2)/2$, and $(3\boldsymbol{v}_1 - \boldsymbol{v}_2)/2$, which are calculated with ordinary vector operations.

Mutation: Offsprings are mutated by randomly assigning values to 0.5 elements of each solution vector.

Since these genetic operators preserve the variety of populations for a long period, they result in a slow convergence. Therefore, in addition to the population size n, the actual Chorus constraint solver takes another parameter that specifies the maximum limit of possible generations.

Modifying Constraint Systems

In user interface applications, it is necessary for constraint solvers to support the modification of constraint systems, because such applications usually require adding new constraints or removing existing ones to alter geometric structures. Also, to realize interactive and animating applications, solvers should provide edit constraints that repeatedly update values of variables; they are typically used to change object positions for mouse dragging and animation.

Importantly, solvers should be predictable to users [21]; that is, in re-solving new constraint systems, they must make new solutions as similar to previous ones as possible. Otherwise, they will surprise or confuse users by drastic changes of resulting diagrams.

Fortunately, new global optimal solutions usually exist near previous ones. Therefore, we may usually start from previous optimal solutions and search for new local optimal ones only by using a numerical optimization technique.

However, this method is sometimes insufficient since possibly new global optimal solutions do not lie close to previous ones. Our algorithm copes with such situations by internally creating `very weak` stay constraints that try to preserve previous variable values. Including such `very weak` constraints, the algorithm first searches for new global optimal solutions with the genetic algorithm. After that, it excludes the `very weak` constraints and then applies numerical optimization to get more exact final solutions.

THE MODULE MECHANISM

The constraint satisfaction algorithm in the previous section models each arithmetic constraint as an error function $e(\boldsymbol{x})$, which allows us to express arithmetic constraints as the program to compute $e(\boldsymbol{x})$ and its gradient $\nabla e(\boldsymbol{x})$. Thus the constraint solver can separately give actual meanings to constraints by introducing evaluation modules to compute such functions. Also, for non-arithmetic constraints, we construct an evaluation module that computes their error function.

We separate numerical optimization programs for local

[2] When we use the C++ version of Chorus on the x86 platform such as Pentium III, we can adopt hardware support for the 80-bit long double precision. In this case, we employ the larger weights 64^4, 64^3, 64^2, 64^1, and 1 instead.

searches as replaceable optimization modules. As stated earlier, programs for required linear equality constraint processing and the genetic algorithm can be implemented independently of numerical optimization.

With these modularizations, the constraint solver is structured as depicted in Figure 2. Object `solver`, the main body of the solver, contains multiple evaluation modules and a single optimization module `optimizer`.

The constraint solver works as follows: When receiving a constraint, `solver` passes it to an appropriate evaluation module according to the strength and kind of the constraint (it directly handles required linear equality constraints without using evaluation modules). For example, it will pass a `medium` linear constraint to `mediumLinearArithmeticEvaluator`. When actually solving constraints, `solver` processes required linear equality constraints and performs global searches, in which it asks `optimizer` to do local searches. Then `optimizer` executes numerical optimization by repeatedly calling evaluation modules to compute an objective function.

Constraints and Evaluation Modules

Constraints are defined by inheriting from class `Constraint` as illustrated in Figure 3(a). Evaluation modules inherit from class `Evaluator` as shown in Figure 3(b). Each evaluation module holds a list `constraints` of its own constraints, and also defines methods `evaluate()` and `gradient()` for combining all the error functions and gradients respectively.

Arithmetic constraints. Class `ArithmeticConstraint` implements arithmetic constraints by specifying an error function $e(\boldsymbol{x})$ and its gradient $\nabla e(\boldsymbol{x})$ as methods `evaluate()` and `gradient()`.

Evaluation modules belonging to class `ArithmeticEvaluator` process arithmetic constraints. The `evaluate()` and `gradient()` methods of this class call `evaluate()` and `gradient()` of its constraints and sum up the results. In Figure 2, the evaluation module `strongArithmeticEvaluator` is an instance of `ArithmeticEvaluator`, and processes `strong` arithmetic constraints.

To alleviate the problem with the approximation of constraint hierarchies, we restrict `medium` and `weak` constraints to being linear (which we will discuss later). We use `LinearArithmeticEvaluator` for these strengths by introducing it as a subclass of `ArithmeticEvaluator` and limiting it to linear constraints.

Non-arithmetic constraints. We permit non-arithmetic constraints to be `strong` only. Unlike `ArithmeticConstraint`, classes for non-arithmetic constraints do not provide the `evaluate()` and `gradient()` methods but simply hold information specific to individual constraints. Instead, evaluation modules for non-arithmetic constraints compute `evaluate()` and `gradient()` by using such constraint-specific information.

We currently present `GraphLayoutConstraint` as non-arithmetic constraints, and `GraphLayoutEvaluator` as their evaluation module to realize graph layouts based on the spring model [15]. Each instance of `GraphLayoutConstraint` indicates a graph edge and holds information about which pair of point variables it connects, which allows programmers to specify each edge as a constraint.

Implicit constraints. In addition to explicit constraints given externally, there are implicit ones that Chorus creates internally. Such implicit constraints are classified into two kinds: variable domain constraints, $\alpha_i \leq x_i \leq \beta_i$, for describing the possible value ranges $[\alpha_i, \beta_i]$ of variables x_i, and `very weak` stay constraints described in the previous section. Implicit constraints are not created as objects. If the used numerical optimization technique for local searches can actually handle required linear inequality constraints, it processes variable domain constraints as required ones; otherwise, an evaluation module instance of `VariableDomainConstraintEvaluator` realizes them as `very strong` constraints by using the error functions

$$e_i(\boldsymbol{x}) = (\max\{0, \alpha_i - x_i, x_i - \beta_i\})^2.$$

Similarly, `DefaultStayConstraintEvaluator` implements `very weak` stay constraints.

Optimization Modules

Optimization modules implement numerical optimization techniques for local searches stated in the previous section. They are defined by inheriting from class `Optimizer` and implement the method `optimize()`. As stated earlier, whether gradients of error functions are used in `optimize()` depends on such numerical techniques.

We currently provide two optimization modules utilizing gradients. `QuasiNewtonOptimizer` performs numerical optimization using the quasi-Newton method based on Broyden-Fletcher-Goldfarb-Sahnno updating formula [4]. As described earlier, the quasi-Newton method only optimizes an objective function and does not handle required constraints. Therefore, required constraints other than linear equations (which are not processed by `solver`) are approximately treated as preferential, `very strong` constraints.

Optimization module `Donlp2Optimizer` exploits a nonlinear programming library DONLP2 [26], which optimizes an objective function subject to required nonlinear constraints. This module processes required constraints other than linear equations by passing them to DONLP2.

IMPLEMENTATION

We have implemented three versions of the Chorus constraint solver: a C++ version, a pure Java version, and a Java version

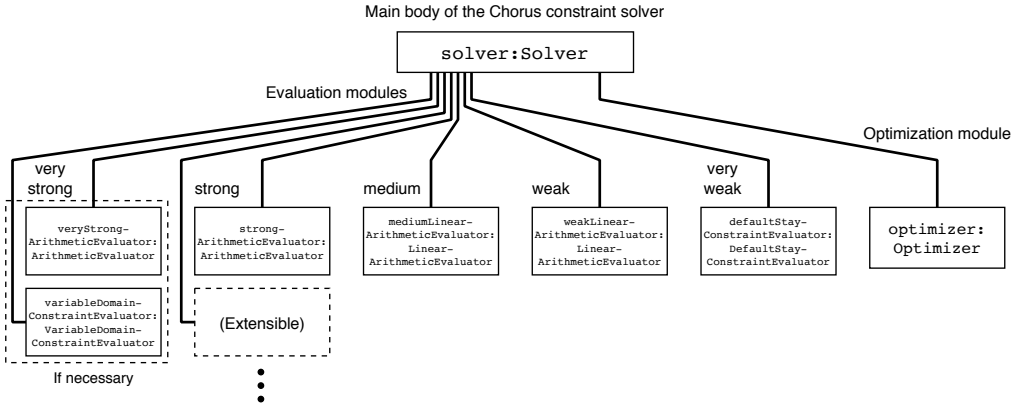


Figure 2: The object structure of the Chorus constraint solver, where `solver` is the external object, and contains the evaluation and optimization modules (`object:Class` means that `object` is an instance of `Class`).

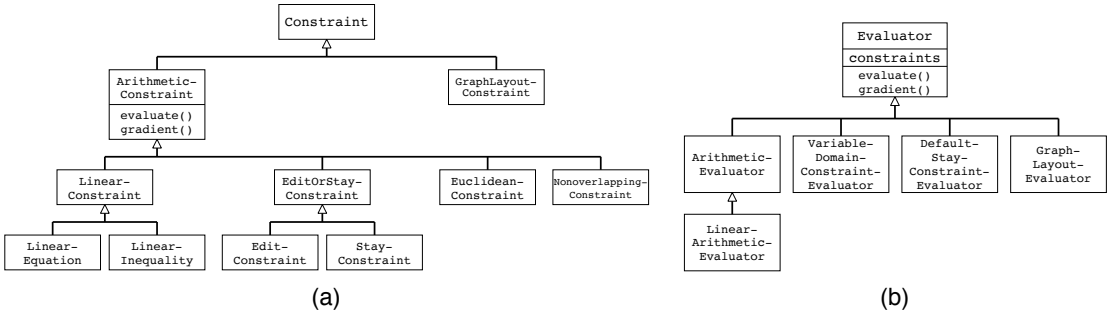


Figure 3: The class hierarchies of (a) constraints and (b) evaluation modules.

using native methods that invoke the C++ implementation. The present C++ implementation consists of approximately seven thousand lines of code.

As shown in Figure 4, we have developed sample applications by using Chorus. We implemented them in C++ with Gimp Toolkit (GTK+), and can compile and execute them on both UNIX and Microsoft Windows. All of these applications provide fully interactive interfaces that allow users to drag objects such as vertices, boxes, and graph nodes in real time on commodity personal computers.

EXPERIMENTS

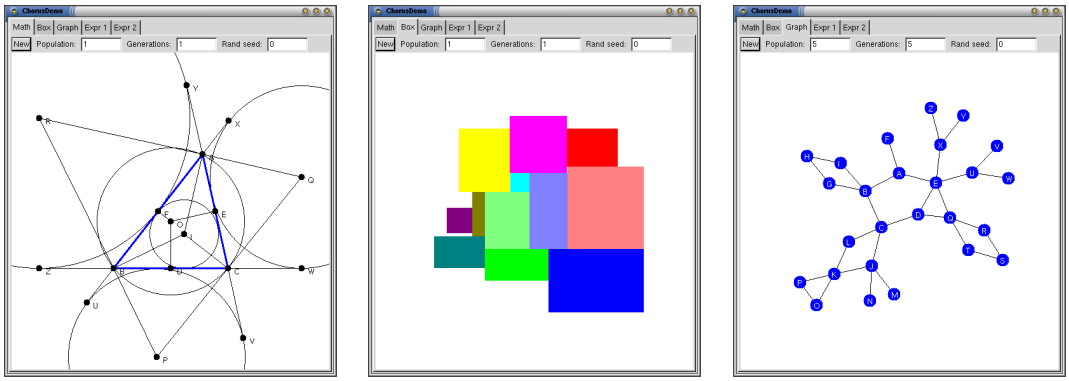
As an evaluation of the methods presented in this paper, we provide the results of experiments on the performance of the Chorus constraint solver in C++. We compiled programs with GNU C EGCS-2.91.66 using the –O3 optimization option, and executed them on a personal computer with an 800 MHz Pentium III processor running Linux 2.2.14.

In the first experiment, we used the actual applications in Figure 4. We measured the times for constraint satisfaction by using `QuasiNewtonOptimizer`. While the mathematical diagram in Figure 4(a) and the box layout in Figure 4(b) do not necessitate global searches, the graph layout in Figure 4(c) requires a global search to obtain an initial solution.

The mathematical diagram in Figure 4(a) consists of 39 variables (for 17 points and 5 radii), 12 `required` linear equality constraints (6 for putting midpoints on triangle sides and 6 for giving initial positions to triangle vertices), 7 `strong` Euclidean geometric constraints (for describing inscribed, circumscribed, and escribed circles), and 6 `weak` stay constraints (for vertex positions). The average computation time for ten trials of local searches for initial layouts was 187 milliseconds. A single update for dragging a vertex was finished typically within 30 milliseconds.

The box layout in Figure 4(b) uses 48 variables (for the opposite vertex pairs of 12 boxes), 24 `required` linear equality constraints (for fixing box sizes), 66 `strong` nonoverlapping constraints, and 24 `weak` stay constraints (for box positions). The average time for ten trials of local searches for initial layouts was 328 milliseconds. Each computation for dragging a box (which possibly caused other boxes to move) was done usually within 20 milliseconds.

The graph layout application in Figure 4(c) contains 52 variables (for 26 nodes) and 31 graph layout constraints. When it was given the population size 10 and the maximum genera-



(a) (b) (c)

Figure 4: Applications of the Chorus constraint solver: (a) a diagram of a triangle and its inscribed, circumscribed, and escribed circles; (b) a non-overlapping box layout; (c) a general graph layout.

tion limit 10, the average time for ten trials of global searches for initial layouts was 4,112 milliseconds. It obtained each dragged layout typically within 50 milliseconds.

In the second experiment, we examined the effect of global searches based on the genetic algorithm by using the application in Figure 5. This application finds a ladder-shaped graph layout consisting of a series of rectangles. Computing this layout requires a global search, because it tends to fall into a local optimal solution due to the twists of the ladder shape. We must note that although this experiment illustrates the effect of the genetic algorithm, it does not guarantee that the algorithm always obtains global optimal solutions.

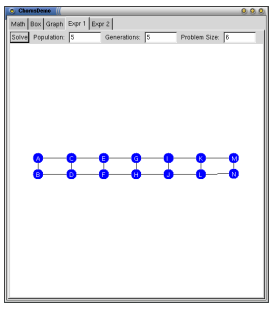


Figure 5: The application for the experiment on the genetic algorithm.

The actual experiment used a layout with 12 rectangles, assigned the population size to 10 and the generation limit to 10, and adopted `QuasiNewtonOptimizer` for local searches. Figure 6 shows how the values of the objective function $E(\boldsymbol{x})$ changed during ten trials of global searches. Each chart represents a trial, and in a chart, each generation plots ten points corresponding to potential solutions. These points are intentionally shifted right and left to be identified even when distinct potential solutions exhibit equal values of the objective function.

All the ten trials obtained global optimal solutions that correspond to graph layouts with no twists. The average computation time was 10,455 milliseconds. As shown in these charts, global optimal solutions were rarely found initially, but potential solutions gradually converged into the global optimality as generations progressed.

In the final experiment, we compared the genetic algorithm with simulated annealing, which is another stochastic algorithm for global searches. We examined the quality of solutions obtained by once executing simulated annealing in the same condition as the second experiment. In this experiment, we exploited the implementation called Simulated Annealing Package [3]. The average time for ten trials was 748 milliseconds. Among the ten trials, only one trial actually found a global optimal solution, and the average number of twists in the final graph layouts was 2.1. However, it might be possible to obtain a better result by using another implementation of simulated annealing.[3]

RELATED WORK AND DISCUSSION

Local propagation constraint solvers [5, 25] can be regarded as a method that realizes a module mechanism, because local propagation constraints can be defined as procedures or functions in ordinary programming languages. However, local propagation solvers impose a limitation that they can hardly deal with simultaneous constraints and inequalities.

Similarly to Chorus, constraint solvers based on numerical optimization have been proposed for a few years [2, 13, 19]. In particular, QOCA [19] enables the modularization of constraint satisfaction algorithms and supports multiple criteria for constraint hierarchy solutions. However, all of them are specialized in linear equality and inequality constraints.

Sketchpad [27] and ThingLab [1] satisfy simultaneous (possibly nonlinear) constraints by switching to a numerical

[3] Although we also tried another implementation SIMANN [9], it imposed a problem with the accuracy of the local optimality of solutions.

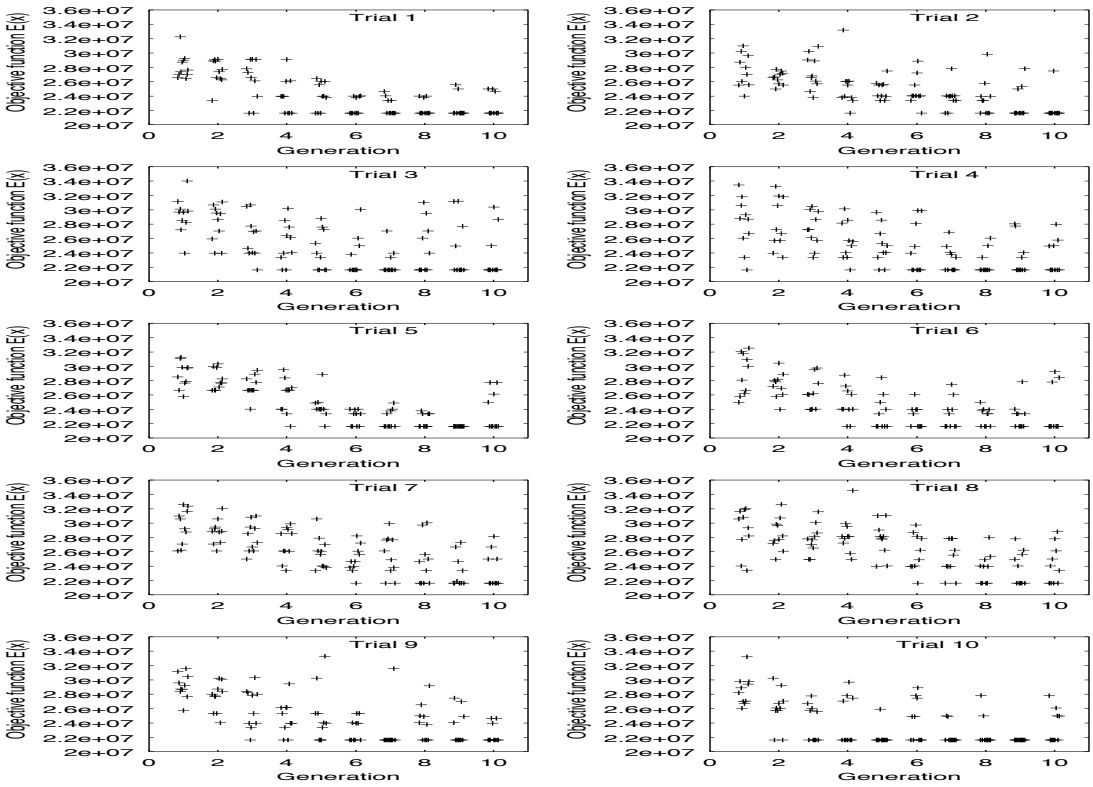


Figure 6: The experimental results of the genetic algorithm.

method called relaxation when they fail in local propagation. Juno [21] and Juno-2 [12] use Newton's method to solve nonlinear geometric constraints. However, they do not provide soft constraints, and are almost incapable of handling inequality constraints.

TRIP [16] and the system in [10] execute Newton's method to find graph layouts based on the spring model [15]. TRIP supports two-level hierarchies of linear equality constraints, but does not allow graph layout and linear equality constraints simultaneously. Although the latter system permits graph layouts with soft linear constraints, it does not enable other nonlinear constraints.

Geometric constraint solvers have also been studied in the field of computer-aided design [7, 18, 22]. They typically handle Euclidean geometric constraints on points and lines (see [24] for samples). A major approach in this field is to satisfy constraints in a "constructive" fashion by appropriately placing geometric objects step by step according to the analysis of constraint systems. Although it achieves fast and accurate constraint satisfaction, it usually excludes soft constraints and inequalities, and also is not applicable to other kinds of constraints such as graph layout constraints.

The Bramble [8] and GLIDE [23] systems realize constraint satisfaction by running virtual dynamic simulations. They allow users to interactively affect constraint satisfaction by using mice; in other words, the users can help them move to more globally optimal states. This dynamic approach might be more appropriate to some interactive applications than using a genetic algorithm which would be slow. However, the dynamic approach does not conform to the ordinary view of constraints; in this approach, constraints must be defined as forces among objects.

To solve geometric constraints, a few attempts to adopt genetic algorithms have been done [6, 20]. To our knowledge, the previous attempts have not achieved practical constraint solvers like Chorus.

Chorus has limitations inherent in its algorithm. To search for global optimal solutions, it adopts a genetic algorithm which is sometimes slow for interactive applications. Fortunately, we can usually realize interactive operations such as dragging without the genetic algorithm. Also, we can easily reduce the cost of the genetic algorithm by making its population size and generation limit smaller, although this may result in less accurate solutions as a tradeoff.

Chorus solves constraint hierarchies approximately, which sometimes produces problems in practice; unlike constraint hierarchies, it may be erroneously affected by weak constraints conflicting with stronger ones. While this kind of approximation can also be found in QOCA, the approximation of Chorus is rougher due to its nonlinearity. Particularly, the

approximation may become worse when both bounded and unbounded error functions are used simultaneously; a weak constraint with an unbounded error function may seriously influence a stronger constraint with a bounded function. To alleviate such problems, we decided to restrict Chorus to having only linear constraints at the `medium` and `weak` levels. This restriction is based on our observation that, in constraint hierarchies, strong constraints are typically used to describe structures of geometric layouts while weaker ones are defined as edit and stay (i.e., linear) constraints to give hints for sizing and positioning the layouts. If `medium` and `weak` constraints are used only in such a way, we can more easily foresee problematic situations caused by the approximation. Of course, this approach is not sufficiently satisfactory, and we are pursuing a more substantial solution.

CONCLUSIONS AND FUTURE WORK

In this paper, we proposed the Chorus constraint solver, which realizes various powerful nonlinear geometric constraints such as Euclidean geometric, non-overlapping, and graph layout constraints. Chorus provides a module mechanism that allows users to define new kinds of geometric constraints, and also supports soft constraints with hierarchical strengths.

As future work, we are planning to enhance the scalability of constraint satisfaction and also to improve the approximation of constraint hierarchies. Other future directions are to develop evaluation modules for more advanced graph layout methods, and also to support three-dimensional applications.

REFERENCES

1. Borning, A. The programming language aspects of ThingLab, a constraint-oriented simulation laboratory. *ACM Trans. Prog. Lang. Syst. 3*, 4 (1981), 353–387.
2. Borning, A., Marriott, K., Stuckey, P., and Xiao, Y. Solving linear arithmetic constraints for user interface applications. In *Proc. ACM UIST*, 1997, 87–96.
3. Carter, E. Simulated Annealing Package. http://www.taygeta.com/annealing/ .
4. Fletcher, R. *Practical Methods of Optimization*, 2nd ed. John Wiley & Sons, 1987.
5. Freeman-Benson, B. N., Maloney, J., and Borning, A. An incremental constraint solver. *Commun. ACM 33*, 1 (1990), 54–63.
6. Frick, A., Keskin, C., and Vogelmann, V. Integration of declarative approaches. In *Graph Drawing—GD'96*, vol. 1190 of *LNCS*, Springer, 1997, 184–192.
7. Fudos, I. *Constraint Solving for Computer Aided Design*. PhD Thesis, Dept. of Computer Science, Purdue University, 1995.
8. Gleicher, M. A graphical toolkit based on differential constraints. In *Proc. ACM UIST*, 1993, 109–120.
9. Goffe, B. SIMANN. Netlib. http://www.netlib.org/opt/simann.f .
10. He, W., and Marriott, K. Constrained graph layout. In *Graph Drawing—GD'96*, vol. 1190 of *LNCS*, Springer, 1997, 217–232.
11. Herrera, F., Lozano, M., and Verdegay, J. L. Tackling real-coded genetic algorithms: Operators and tools for behavioural analysis. *Artif. Intell. Rev. 12*, 4 (1998), 265–319.
12. Heydon, A., and Nelson, G. The Juno-2 constraint-based drawing editor. Research Report 131a, Digital Systems Research Center, 1994.
13. Hosobe, H. A scalable linear constraint solver for user interface construction. In *Principles and Practice of Constraint Programming—CP2000*, vol. 1894 of *LNCS*, Springer, 2000, 218–232.
14. Igarashi, T., Matsuoka, S., Kawachiya, S., and Tanaka, H. Interactive beautification: A technique for rapid geometric design. In *Proc. ACM UIST*, 1997, 105–114.
15. Kamada, T., and Kawai, S. An algorithm for drawing general undirected graphs. *Inf. Process. Lett. 31*, 1 (1989), 7–15.
16. Kamada, T., and Kawai, S. A general framework for visualizing abstract objects and relations. *ACM Trans. Gr. 10*, 1 (1991), 1–39.
17. Kitano, H., Ed. *Genetic Algorithms*. Sangyo-Tosho, 1993. In Japanese.
18. Kramer, G. A. A geometric constraint engine. *Artif. Intell. 58*, 1–3 (1992), 327–360.
19. Marriott, K., Chok, S. C., and Finlay, A. A tableau based constraint solving toolkit for interactive graphical applications. In *Principles and Practice of Constraint Programming—CP98*, vol. 1520 of *LNCS*, Springer, 1998, 340–354.
20. Masui, T. Graphic object layout with interactive genetic algorithms. In *Proc. IEEE VL*, 1992, 74–80.
21. Nelson, G. Juno: A constraint-based graphics system. In *Proc. ACM SIGGRAPH*, 1985, 235–243.
22. Owen, J. C. Algebraic solution for geometry from dimensional constraints. In *Proc. ACM Solid Modeling*, 1991, 397–407.
23. Ryall, K., Marks, J., and Shieber, S. An interactive constraint-based system for drawing graphs. In *Proc. ACM UIST*, 1997, 97–104.
24. Saltire Software. Constraint geometry. http://www.saltire.com/constraints.html .
25. Sannella, M. SkyBlue: A multi-way local propagation constraint solver for user interface construction. In *Proc. ACM UIST*, 1994, 137–146.
26. Spellucci, P. DONLP2. Netlib. http://www.netlib.org/opt/donlp2/ .
27. Sutherland, I. E. Sketchpad: A man-machine graphical communication system. In *Proc. AFIPS Spring Joint Conf.*, 1963, 329–346.
28. Tamassia, R. Constraints in graph drawing algorithms. *Constraints J. 3*, 1 (1998), 87–120.