A Geometric Constraint Solver

William Bouma* Ioannis Fudos† Christoph Hoffmann*
Department of Computer Science, Purdue University
West Lafayette, IN 47907-1398

Jiazhen Cai‡ Robert Paige‡
Department of Computer Science, Courant Institute
251 Mercer Str., New York, NY 10012

Report CSD-TR-93-054§

Abstract

We report on the development of a two-dimensional geometric constraint solver. The solver is a major component of a new generation of CAD systems that we are developing based on a high-level geometry representation. The solver uses a graph-reduction directed algebraic approach, and achieves interactive speed. We describe the architecture of the solver and its basic capabilities. Then, we discuss in detail how to extend the scope of the solver, with special emphasis placed on the theoretical and human factors involved in finding a solution — in an exponentially large search space — so that the solution is appropriate to the application and the way of finding it is intuitive to an untrained user.

1 Introduction

Solving a system of geometric constraints is a problem that has been considered by several communities, and using different approaches. For example, the symbolic computation community has considered the general problem, in the

*Supported in part by ONR contract N00014-90-J-1599, by NSF Grant CDA 92-23502, and by NSF Grant ECD 88-03017.

†Supported by a David Ross fellowship.

‡Supported in part by ONR contract N00014-90-J-1890, by AFOSR grant 91-0308, and by NSF grant MIP 93-00210.

§Revised January 1994. This report and others are available via anonymous ftp to arthur.cs.purdue.edu, in directory pub/cmh and subsidiaries.

context of automatically deriving and proving theorems from analytic geometry, and applying these techniques to vision problems; [5, 8, 14]. The geometric modeling community has considered the problem for the purpose of developing sketching systems in which a rough sketch, annotated with dimension and constraints, is instantiated to satisfy all constraints. This work will be reviewed in the next section. The applications of this approach are in mechanical engineering, and, especially, in manufacturing.

With this work, we have mainly manufacturing applications in mind. Our purposes and goals are as follows:

1. We develop a constraint solver in which the information flow between the user interface and the underlying solver has been formalized by a high-level representation that is minimally committed to the particular capabilities or technical characteristics of the solver, and is independent of the interface. Such a representation can become the basis for archiving sketches in a neutral format, with the ability to retrieve the archived sketch and edit it; [13, 12]. Our solution is also a building block for a larger project of developing a new generation of CAD systems based on a neutral, high-level geometry representation that expresses design intent and preserves the ability to redesign.

2. We explore the utility of several different general-purpose and interoperating rapid prototyping languages and systems for developing specific tools for experimenting conveniently with a variety of ideas and approaches to constraint solving. Aside from well-known special purpose tools such as LEX and Yacc, our constraint solver also makes use of the high level language SETL2 [27] to specify complex combinatorial algorithms and the transformational system APTS [7, 23] to perform syntactic analysis and symbolic manipulation of geometrical constraint specifications.

3. We study a number of neglected aspects of constraint solving. They include

   (a) redirecting the solver to a different solution of a well-constrained sketch,

   (b) devising generic techniques for extending the capabilities of the solver while preserving interactive speed, and

   (c) a rigorous correctness proof of the solver.

   Note that the correctness proof is reported separately [10].

This paper reports substantial progress in all three problem dimensions, and identifies a number of open issues that remain.

2 Approaches to Geometric Constraint Solving

We consider only well-constrained, two-dimensional sketches formed from points, lines, circles, segments and arcs. Constraints are explicit dimensions of distances and angles, as well as constraints of parallelism, incidence, perpendicularity, tangency, concentricity, collinearity, and prescribed radii. We exclude relations on dimension variables and inequality constraints. In particular, the user specifies a rough sketch and adds to it geometric and dimensional constraints that are normally not yet satisfied by the sketch. The sketch only has to be topologically correct. The constraint solver determines from the sketch the geometric elements that are to be found, and processes the constraints to determine each geometric element such that the constraints are satisfied.

Solving over- and underconstrained sketches is of significant practical interest. In our experience, solving underconstrained sketches is customarily approached by adding, as the solution progresses, constraints derived from the metric properties of initial sketch, using ad-hoc rules for the selection of such constraints. We can prove that an underconstrained sketch can be partially solved in a unique way, using the given constraints; [10]. However, adding constraints deduced from the metric properties of the user-supplied rough sketch must rely on a heuristic selection. We explain in Section 5 why this is difficult.

We also argue in Section 5 that consistently over-constrained sketches have the attractive property that the number of possible solutions is reduced. In consequence, such constraint problems can unambiguously specify user-intent. However, finding a solution is intractable even for very simple configurations, and more research is needed to devise demonstrably effective approaches.

Our constraint solver is *variational*. That is, the solver is not obliged to process the constraints in a predetermined sequence, and the constraints specified by the user are not parametric in the sense that they must be determined serially, each as an explicit function of the previous constraints. This is analogous to writing the constraints in a *declarative* language, where the solution is independent of the order in which the constraints are written down. This greatly increases the generality of the constraint solving problem, and demands solvers that are based on advanced mathematical concepts.

While users of geometric constraint solving systems think geometrically and express themselves with visual gestures, constraint solvers work with a different internal representation. Most users will be quite unaware of the nature of the underlying representation and of the internal workings of the constraint solver. Coupled with the fact that a well-constrained geometric constraint problem has, in general, exponentially many solutions, only one of which satisfies the user's intent, constraint solvers therefore have to address two distinct tasks:

1. Determine whether the problem can be solved and if so, how.

2. Among the possible solutions, identify the one the user has intended.

Most of the literature assumes tacitly that the second task is easy to discharge. In Section 5, we question this assumption and show why Task 2 is difficult.

Before describing our approach to Task 1, we characterize other approaches in the literature. Many constraint solvers use a combination of these methods. To satisfy editorial requirements, our review is very brief. For a more detailed analysis of the extensive prior work on constraint solving see [9].

2.1 Numerical Constraint Solvers

In numerical constraint solvers, the constraints are translated into a system of equations and are solved using an iterative method. When based on Newton iteration, such solvers require good initial values, so that the initial sketch must almost satisfy all constraints already. Such solvers are quite general, and are capable of dealing with overconstrained, consistent constraint problems. Examples include Solano and Brunet [29], and Gossard and Light [21].

Nonlinear systems have an exponential number of solutions, but Newton iteration will find only one. Numerical solvers are therefore inappropriate when the initial sketch is only topologically correct, or when the solver locks into a solution that is unsuited to the application and has no method with which to find more suitable alternatives.

Sutherland's Sketchpad [31] was the first system to use the method of numerical relaxation. Many systems since then can do relaxation as an alternative to some other method.

2.2 Constructive Constraint Solvers

This class of constraint solvers is based on the fact that most configurations in an engineering drawing are solvable by ruler, compass and protractor, or using another, less classical repertoire of construction steps. In these methods, the constraints are satisfied constructively, by placing geometric elements in some order that may or may not be fixed. This is more natural for the user and makes the approach suitable for interactively debugging a sketch.

2.2.1 Rule-Constructive Solvers

Rule-constructive solers use rewrite rules to discover and execute the construction steps. Bruderlin and Sohrt [3, 28] solve constraints in this way and incorporate the Knuth-Bendix critical-pairs algorithm [16]. They show that their method is correct and solves all problems that can be constructed using ruler and compass. Other rule-constructive solvers include Aldefeld [1] and Sunde [30]. In [35] and in [33] the problem of nonunique solutions is considered.

Although a Logic Programming style of constraint solving is a good approach for prototyping and experimentation, the extensive computations searching and matching rewrite rules constitute a liability.

2.2.2 Graph-Constructive Solvers

Graph-constructive solvers have two phases. First, a graph representing the constraints is analyzed and a sequence of construction steps is derived. Second, the construction steps are carried out to derive the solution.

This approach is fast and more methodical than the rule-constructive approach. However, as the repertoire of possible constraints increases, the graph-analysis algorithm has to be modified.

Requicha [26] uses dimensioned trees that allow only horizontal and vertical distances. Todd [32] generalizes the dimension trees and gives a characterization of the expressive power of the solver. In [22], Owen presents an extension of this principle to include circularly dimensioned sketches, and DCM is a commercial constraint solver using this method.

Since our core algorithm is similar to [22], we describe Owen's solvers in more detail. Owen analyzes the constraint graph for triconnected components. Each triconnected component is reduced to a number of elements that interact with other components, and a determination is made how these elements fit together. Thereafter, each component is separately determined. The procedure is recursive in that once components have been reduced, they in turn can become members of triconnected components in the reduced graph.

A key aspect of the solver is that only ruler-and-compass construction construction steps are allowed. Algebraically, this is equivalent to solving only quadratic equations, so that the specific coordinate computations do not require sophisticated mathematical computations. [22] argues that the solver is complete for ruler-and-compass constructible point configurations with prescribed distances that are algebraically independent.

Kramer [19] describes a 3D constraint solver that deals with constraints from kinematics that are characterized by basic joint types. Complex geometric elements are placed implicitly by choosing a suitable number of local coordinate frames to be placed with respect to each other by rigid-body motions.

2.3 Propagation Methods

Constraint propagation was a popular approach in early constraint solving systems. The constraints are first translated into a system of equations involving variables and constants, and an undirected graph is created whose nodes are the equations, variables and constants, and whose edges represent whether a variable or constant occurs in an equation. The method then attempts to direct the graph edges so that every equation can be solved in turn, initially only from the constants. To succeed, various propagation techniques have been tried, but none of them is guaranteed to derive a solution when one exists, and most fail when presented with a circularly constrained problem. For a detailed review see [20, 28].

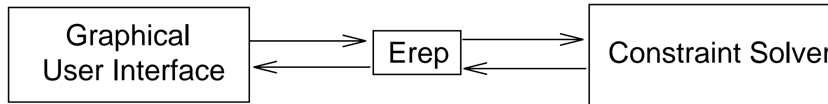

Figure 1: Architecture of the Constraint Solver

2.4 Symbolic Constraint Solvers

The constraints are translated into a system of algebraic equations. The system is solved with symbolic algebraic methods, such as Gröbner's bases, e.g., [5, 11], or the Wu-Ritt method, e.g., [34, 8]. Both methods can solve general nonlinear systems of algebraic equations, but may require exponential running times.

In [17, 18], Kondo considers the addition and deletion of constraints using the Buchberger's Algorithm [5] to derive a polynomial that gives the relationship between the deleted and added constraints.

3 The Constraint Solving System

3.1 Information Flow and Rationale

The overall architecture of the constraint solver is shown in Figure 1. The user draws a sketch and annotates it with geometric constraints. Additional capabilities include interacting with the solver to identify a different, valid solution.

The user interface translates the specification into a textual language that records the problem faithfully. The user could edit this textual problem specification, but this is unnecessary, because the specification is edited and updated automatically by the user interface. The language has been designed to achieve the objectives of [13] — a neutral problem specification that makes no assumptions about the architecture of the underlying constraint solving algorithm. Thus, it is quite easy to federate any constraint solver capable of handling the geometric configurations we consider.

The textual problem specification is handed to the constraint solver which translates the constraints into a graph, and, as described later, solves them by graph reductions that govern the workings of our algebraic, variational constraint solver. The solver capabilities are the consequence of certain construction steps that have been implemented. If a particular constraint problem can be solved using these steps, then our solver will find a solution. Where the construction steps involve ruler-and-compass constructions, only quadratic equations need to be solved. But some construction steps are permitted that are not ruler-and-compass, and in those situations the roots of a univariate polynomial are found numerically. This polynomial has been precomputed except for the coefficients which are functions of specific constraint values.

A well-constrained geometric problem can have exponentially many solutions in the number of constraints. This is because the solutions correspond to the algebraic set of a zero-dimensional ideal whose generating polynomials are nonlinear. Our solver can determine all possible solutions. But doing so every time would waste time and overwhelm the user. So, certain heuristics, described later, narrow down the solutions to a final configuration that corresponds to the intended solution with high probability. In case a different solution is wanted, the solver can be redirected to the intended solution interactively.

Our system is a component of a constraint-driven variational CAD system based on a high-level, declarative, editable geometry representation (Erep) as discussed in [13, 12]. Such an overall architecture poses several challenges. One of them is efficient variational constraint solving, and we address this problem here. Another, key challenge is to formulate the language in a neutral way, committing it neither to the particulars of the user interface nor of the solver algorithms. This is a more subtle challenge because the way in which dimensions are displayed in the sketch has to make some assumptions about the capabilities of the user interface. Likewise, interacting with the solver to find alternative solutions requires conceptualizing the solution process in a way that makes no assumptions about how they are found. Here, we assume only that the solver is capable of undoing the last placement operation, and can look for a different placement of a geometric element. The textual protocol for communicating these matters is encapsulated.

3.2 System Implementation

The graphical user interface is an X application written in C++ using the Motif widget set. The user-prepared sketch is changed into an Erep specification and is passed as text to the constraint solver.

The solver is written using two novel software tools — the APTS transformational programming system [7, 23] and the high-level language SETL2 [27] — each having special features that the solver exploits. The front-end to the constraint solver engine is an APTS program that reads the Erep program and type checks it. For example, we check that only lines participate in angle constraints. If there are no obvious type errors, the Erep program is transformed into an equivalent Erep specification in a normal form in which only distance and angle constraints are allowed. For example, incidence constraints are translated to zero-distance constraints. The specification of the orientation of lines in angle constraints is also regularized. Relations representing a constraint graph are then extracted from the Erep program and are exported via a foreign interface to a SETL2 program that implements the main algorithmic part of the solver.

The use of such novel systems as APTS and SETL2 is motivated by the special needs of our project. A major part of our research is the discovery and implementation of complex, nonnumerical algorithms. Our goal of high

performance based on a new algebraic approach to constraint solving entails deep graph-theoretic analysis of implicit dependencies between constraints, and complex graph traversals based on such analysis. A wide variety of techniques seem available to us, but proper evaluation requires extensive labor-intensive computational experiments. Using these tools has allowed us to implement our algorithms with surprising speed. In the future we also hope to make use of a promising new technology for mechanically transforming prototype SETL2 programs into high performance C code [7].

The special syntactic, semantic, and transformational capabilities of APTS are also well suited to a flexible, experimental development of a logical framework with an evolving Erep language and corresponding solver. Like systems such as Centaur [2] the Synthesizer Generator [25], and Refine [24], APTS has a single uniform formalism for lexical analysis, syntactic analysis, and pretty-printing. However, the semantic formalism in APTS has several advantages over the more conventional attribute grammar approach [15] that is used in the Synthesizer Generator. APTS uses a logic-based approach to semantics in which semantic rules that define relations are written in a Datalog-like language [6] but with the full expressive power of Prolog. These rules are written independently of the individual grammar productions and without reference to the parse tree structure. They define relations over a rich assortment of primitive and constructed domains, and have the brevity and convenience of unrestricted circular attribute grammars. We are not aware of any implementation that allows a comparable unrestricted circularity.

4 Solver Algorithmics and Extensibility

First, we discuss our basic method for solving geometric constraints. While Owen's solver is top-down, determining first the interaction between clusters of geometric elements, our basic method is bottom-up. This allows us to prove correctness of the basic algorithm, both with respect to the constraint graph analysis and to the subsequent geometric construction; [10]. It also permits systematic extensions of the expressive power of the solver, and makes the algorithm easy to describe, implement, and understand.

We begin in the basic algorithm by placing geometric elements until a cluster has been determined. The construction steps needed are described later. Once a cluster cannot be extended, another cluster is constructed in the same way. Several clusters sharing geometric elements are then coalesced based on some simple rules also described, by a rigid motion of one with respect to the other. Coalesced clusters are again treated as clusters, so the recursive nature of Owen's algorithm is also manifest in our approach. In the basic algorithm, only quadratic equations are solved. Thus, the basic algorithm is restricted to ruler-and-compass constructible configurations.

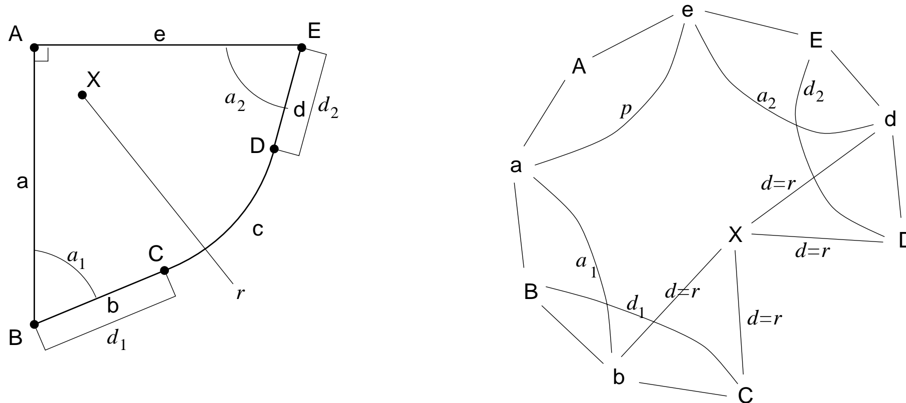

Figure 2: Example Configuration and Corresponding Constraint Graph. Unlabeled edges represent incidence.

For the larger class of geometric elements consisting of points, lines and circles, our basic algorithm and Owen's methods do not solve all ruler-and-compass constructible configurations. For example, for Subcase 1 of Table 1 in Section 4.2.1, our basic solver must be extended. DCM can solve the configuration sometimes, depending on the way the problem is posed. We suspect that a complete ruler-and-compass constructible solver for the larger class of geometric elements requires graph rewriting rules that are equivalent to the Knuth-Bendix algorithm [16].

We also discuss a general method for extending the solver to configurations that cannot be done with the basic algorithm. Our strategy places two clusters related by three constraints. The extension goes beyond ruler-and-compass constructions, and requires a root finder for univariate polynomials. Conceptually, the extension corresponds to adding new geometric construction steps.

4.1 Solving with Graph Reduction

4.1.1 Cluster Formation

The user sketch, annotated with constraints, is translated into a graph whose vertices correspond to geometric elements — points, lines and circles — and whose edges are constraints between them. In particular, a segment is translated into a line and two points, and an arc into a circle, two arc end points, and the center of the circle. For example, the sketch of Figure 2 (left) is translated into the graph of Figure 2 (right). In the graph, d represents a distance constraint, a an angle constraint, and p perpendicularity. Tangency has been expressed by a distance constraint between the center of the circle and the line tangent to the circle. All other graph edges represent incidence. Circles of fixed radius can be determined by placing the center, so there is no vertex corresponding to the

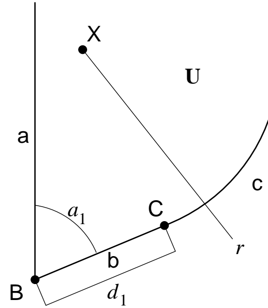

Figure 3: Cluster **U** of Figure 2

circle of the arc c in the constraint graph. The basic idea of the solver algorithm is now as follows:

1. Pick two geometric elements (graph vertices) that are related by a constraint (connected by an edge) and place them with respect to each other. The two elements are now *known*, and all other geometries are *unknown*.

2. Repeat the following: If there is an unknown geometric element with two constraints relating to known geometric elements, then place the unknown element with respect to the known ones by a construction step. The geometric element so placed is now also known.

Note that for subsequent cluster formation one of the two initial geometric elements may already belong to another cluster.

For example, in the graph of Figure 2, we may begin with elements a and B, effectively drawing a line a and placing on it the point B anywhere. We can now place in sequence b, C, and X. At this point, no additional elements can be placed and the cluster is complete, as shown in Figure 3. Note that we neither know where A is situated, nor how far the arc c extends. Starting again, two other clusters are determined. One consists of X, D, d, E, and e. The other cluster consists of a, A, and e. Note that the same geometric element may occur in more than one cluster. Both clusters are shown side-by-side in Figure 4.

4.1.2 Recursion

Two clusters with one geometric element in common and one constraint between them can, in general, be placed with respect to each other. To do so, the shared geometric element is identified, and the remaining degree of freedom eliminated from the additional constraint.

Three clusters, each sharing a geometric element with one of the others, can also be placed relative to each other. By designating the elements v and w in the two-cluster case as a third cluster, the three-cluster placement operation subsumes the two-cluster case. Figure 5 shows both cases.

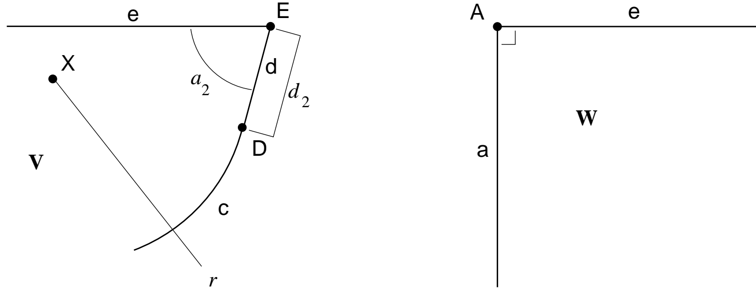

Figure 4: Clusters **V** and **W** of Figure 2

.

Briefly, cluster merging is accomplished as follows: Fix one cluster, say **U**, thereby fixing the two geometric elements u and v. Place w with respect to u and v deriving the needed constraints from the metric properties of clusters **V** and **W**. Now move the two clusters **V** and **W** to match the position of the geometric elements u, v and w, using a rigid-body transformation.

4.1.3 Construction Steps

The reduction steps correspond to standardized geometric construction steps, and also to solving standardized, small systems of algebraic equations. The construction steps include the following:

Basis Steps: The basis steps place two geometric elements related by a graph edge. They include placing a point on a line, placing two lines at a given angle, placing two points at a given distance, and so on. Note that in general there are several ways to place the geometric elements.

Point Placements: These rules place a point using two constraints. They include placing a point at a prescribed distance from two given points, or at prescribed distances from given lines, and so on. See also Figure 6.

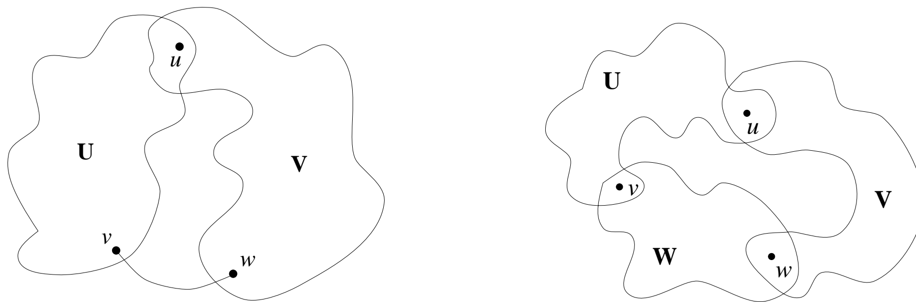

Figure 5: Recursive Cluster Placement

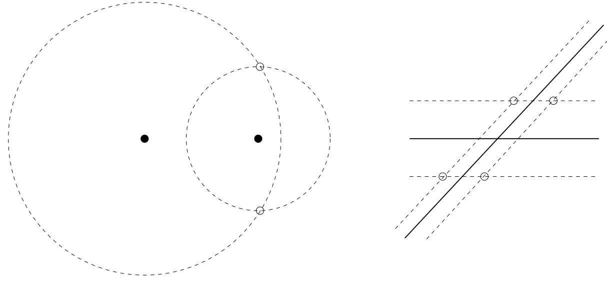

Figure 6: Point Placement Rules: Left, by Distance from Two Points; Right, by Distance from Two Lines.

Line Placements: These rules place a line with respect to two given geometric elements. They include placing a line tangent to a circle through a given point, at given distance from two points, etc.

Circle Placement: These rules place a circle of fixed or variable radius. Fixed-radius circles require only two constraints, and determining them can be reduced to placing the center point. Variable-radius circles require three constraints and reduce in many cases to the Apollonius problem — finding a circle that is tangent to three given ones.

Algebraic Formulation: Geometric elements are represented as follows: Points are represented by Cartesian coordinates. A line is determined from its implicit equation in which the coefficients have been normalized:

$$\mathsf{a} : \; m\,x + n\,y + p = 0 \qquad n^2 + m^2 = 1$$

Consequently, p is the distance of the origin from the line. Because of the normalization, lines are determined only by two numerical quantities, the (signed) distance p of the origin from the line, and the direction angle $\cos\alpha = n$. Therefore, two constraints determine a line. Lines are oriented by choosing $(-n, m)$ as the direction of the line. Circles are represented by the Cartesian coordinates of the center and the radius, an unsigned number.

4.1.4 Graph Transformations

The scope of the basic solver can be extended by certain graph transformations. They are a simple and effective technique to extend the scope of the solver. For example, when two angle constraints α and β are given between three lines, then a third angle constraint can be added requiring an angle of $180° - \alpha - \beta$. We use such transformations.

Note that we avoid transformations that restrict the generality of the solver. For example, consider further constraining the configuration shown in Figure 7 so that point A is on line a, and point C on line c. The situation implies that either lines a and c are incident, or that points A and C are incident. The two

Figure 7: Indidence of A with a and C with c and Resulting Constraint Graph.

possibilities lead to different solutions. If we were to apply a transformation to the constraint graph that added one of the incidences as a new graph edge, then we would have excluded the other possibility, and with it some solutions. If we added both incidence edges, then we would have introduced the unwarranted assumption that both the points and the lines coincide. In each case we can exhibit examples in which a solvable constraint problem becomes unsolvable. This is an example of an undesirable graph transformation.

4.2 Solver Extensions

The basic algorithm for solving constraints given before can be extended to handle more complex geometric situations. The strategy discussed here generalizes the placement of two clusters with respect to each other, when three constraints between them are given.

In constrast to individual geometric elements, clusters have three degrees of freedom. The recursion in the basic algorithm is able to place clusters with shared elements, a very special case. When two clusters are related by three constraints between them, it is possible to place them with respect to each other, but the basic algorithm cannot do so. We now describe how to extend the basic algorithm systematically so that these configurations can be solved. As a simple example of this type of extension, consider Figure 8. A triangle has been specified by givbing the length d of side c, the angle γ, and the height h. As we can see from the associated constraint graph, two clusters must be placed in relation to each other, one consisting of vertex C and the sides a and b, the

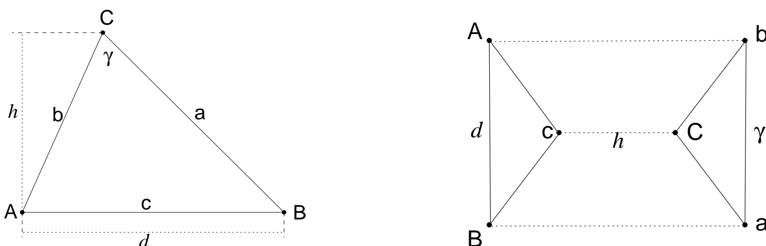

Figure 8: Traingle specified by h, d, and γ. Two clusters are obtained related by three constraints.

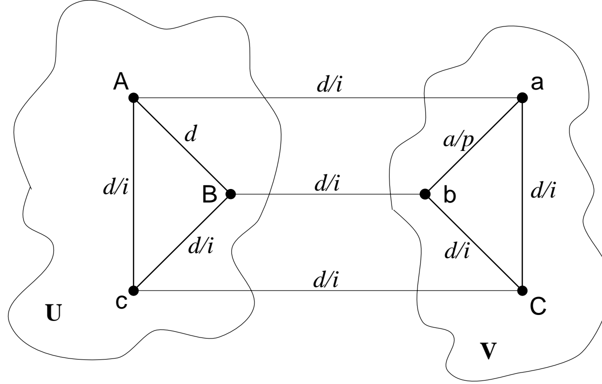

Figure 9: Case $(p, p, l) \equiv (l, l, p)$

other of the side cand the vertices A and B.

Consider the situation shown in Figure 9. A cluster **U** and a cluster **V** have been solved separately, and three distinct elements have been identified in each that are constrained such that the two clusters can be placed with respect to each other. Since six distinct elements are involved, the basic algorithm cannot solve this problem. We use one of the following two conceptual approaches:

(A) We fix one of the clusters. Then, a sequence of ordinary construction steps places the other cluster, possibly with the introduction of auxiliary construction points, lines and/or circles.

(B) Fixing one of the clusters, **V**, we consider how the other cluster, **U**, moves when two of the three constraints are satisfied. **U** can move with one degree of freedom. We precompute the locus of the geometric element in **U** whose constraint has been ignored so far. If this element is a point, its locus is an implicit algebraic curve whose coefficients are expressions in the given constraints. Then, the locus is intersected with a construction line or circle, depending on the nature of the third constraint, and the intersections identify those positions of the third geometric element at which the remaining constraint is also satisfied.

The first method corresponds to a ruler-and-compass construction. The second method is not necessarily equivalent to a ruler-and-compass construction. For example, if both clusters are a triangle and distances between corresponding vertices must be satisfied, then no ruler-and-compass construction exists.

In both methods we conceptually satisfy two constraints and examine the motion under the remaining degree of freedom. Particularly in the case of point loci, classical curves are obtained that are described in the literature. The major cases are distinguished by the possible two-element pairs whose constraints we satisfy a-priori. There are six major cases, two of them involve an angle or

Subcase Number	Constraints Combination	Properties of **U**	Properties of **V**	Method of Solution
1.	A i a c i C	A i c A d B	C i a a a/p b	*(A)*
2.	A i a c i C	A i c A d B	C d a a a/p b	*(B)*
3.	A i a c i C	A d c A d B	C d a a a/p b	*(B)*

Table 1: Essential combinations of $((p,l),(l,p))$. Constraint symbols mean i = incident, d = nonzero distance, p = parallel, a = nonzero angle. The third inter-cluster constraint is B d/i b.

parallel constraint between two lines and can be shown to be solvable by method *(A)*.

Each major case has a number of subcases that depend on the third constraint and on the particular value configurations of the first two constraints. Using suitable coordinate transformations, many of them can be combined. We examine one major case which requires method *(B)* in general, but has subcases that can be solved with method *(A)*.

4.2.1 Case ((p,l),(l,p))

Assume that in cluster **U** the three elements are the points A and B and the line c, and in cluster **V** the elements are two lines a and b and a point C. The constraint possibilities are denoted with d for nonzero distance, i for incidence, a for angle, and p for parallel. In this configuration, we fix cluster **V** and move cluster **U** relative to it such that all constraints are satisfied. We consider how **U** moves when the constraint between A and a and the constraint between C and c are satisfied. Moreover, we focus on the situation in which the constraints between A and a, and between C and c, are incidence constraints. Several subcases arise that have been summarized in Table 1. The other combinations, involving nonincidences, can be mapped to those of the table by replacing some of the lines in **U** and **V** with suitably positioned parallel lines.

Consider Subcase 1, assuming that B has distance t from c, distance r from A, and should have distance e from b. Any of the distances may be zero. The two clusters are shown in Figure 10. Two families of solutions exist: Either the lines a and c coincide, possibly in opposite orientation, or the points A and C

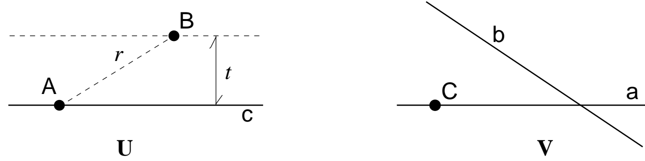

Figure 10: Subcase 1 Configuration

coincide.[1] In the first situation, the locus of B is a pair of lines parallel to a, at distance t. Up to four intersections with lines parallel to b at distance e are possible locations for B, and each of them determines the relative position of **U** with respect to **V**. In the second situation, the locus of B is a circle around C with radius r, and the up to four intersections with lines parallel to b at distance e are possible locations for B. See also Figure 11.

Now consider Subcase 2 in Table 1. We determine the curve that is the locus of B, assuming that B has coordinates (x, y). By a coordinate transformation, moreover, we can assume that C has coordinates $(0, 1)$ and that A, constrained to be on a, has coordinates $(a, 0)$.

In the simplest case, A and B are both on c, distance d_1 apart. In Figure 12 (left) this corresponds to $d_2 = 0$. The cotangent of the angle θ between lines c and a is then $-a$, so that we can express $\sin\theta = 1/u$ and $\cos\theta = -a/u$, where $u = \sqrt{1 + a^2}$. The locus of B can therefore be described by three equations:

$$\begin{aligned} xu - au + d_1 a &= 0 \\ yu - d_1 &= 0 \\ u^2 - a^2 &= 1 \end{aligned} \tag{1}$$

Eliminating a with a Gröbner basis computation establishes that the locus of B

[1] We discussed this configuration in Subsection 4.1.4.

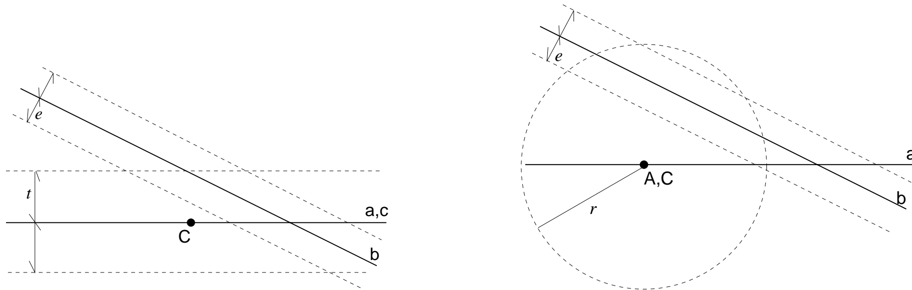

Figure 11: Subcase 1: left, first solution; right, second solution. Locus of B ia a pair of lines parallel to a on the left, and a circle centered at A on the right.

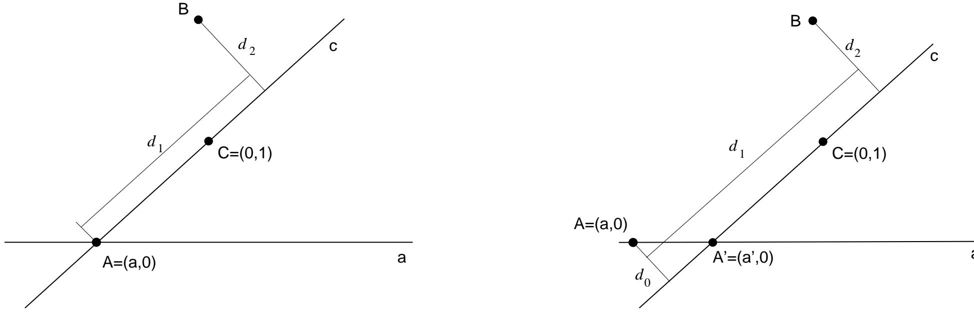

Figure 12: Left: Subcase 2 with B d c, A i c. Right: Subcase 3 with B d c, A d c.

is the degree 4 curve

$$y^4 - 2\,y^3 + x^2 y^2 + y^2(1 - d_1^2) + 2\,d_1^2 y - d_1^2 = 0$$

This curve is intersected with the lines parallel to b at distance e. The up to 8 intersection points determine the possible positions of B.

Now if B is not incident to c, then $d_2 \neq 0$. The equations describing the locus of B are only slightly more complicated, and are

$$\begin{aligned} xu - au + d_1 a + d_2 &= 0 \\ yu - d_1 + d_2 a &= 0 \\ u^2 - a^2 &= 1 \end{aligned}$$

Again, B lies on a degree 4 curve whose coefficients are polynomial in d_1 and d_2 of degree 4. Again this curve is intersected with the lines parallel to b at distance e from which the position of B can be found.

The most general situation is Subcase 3 where A is not on c. Referring to Figure 12 (right), the equations describing the locus of B are

$$\begin{aligned} xu - au + d_1 a' - d_0 + d_2 &= 0 \\ yu - d_1 - d_0 a' + d_2 a' &= 0 \\ a - a' + d_0 u &= 0 \\ u^2 - a'^2 &= 1 \end{aligned}$$

From a Gröbner basis computation one determines that the locus of B is a curve of degree 4. The coefficients are polynomials in the d_k of degree up to 4.

4.2.2 Line Loci

If the element whose locus we determine is not a point, then we need equations for the determining quantities. In the case of lines, those are the coefficients

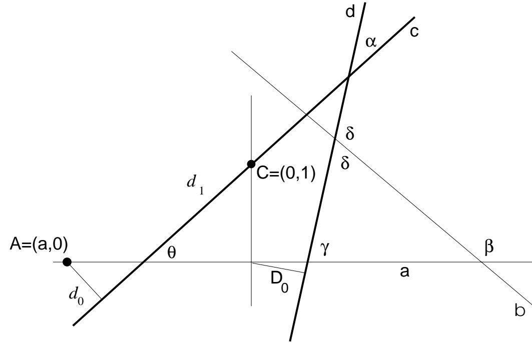

Figure 13: Configuration for Satisfying Line Constraints

of the line equation, or, equivalently, the direction angle and the distance from the origin. For example, consider the case in which the constrained element pairs are (p, l), (l, p), and (l, l). Suppose we approach the situation by satisfying the constraints between the two point-line pairs, and precompute how the line equation varies. Then the subcases are as before. The moving cluster **U** has the point A and the lines c and d, and the fixed cluster **V** has the point C and the lines a and b.

In Subcase 3, the most general situation, we have to determine the distance D_0 of the moving line from the origin and the components of the normal vector after norming it to length 1; see also Figure 13. Let α be the fixed angle between the moving lines c and d, θ the direction angle of c. This angle determines a particular position of the moving cluster. Let γ be the direction angle of line d, and assume that lines d and b should form an angle δ. Let β be the direction angle of b. Then we must solve for θ in

$$\delta = \beta - (\alpha \pm \theta) \qquad \text{or} \qquad \delta = 180 - \beta + (\alpha \pm \theta)$$

accounting for the different positions of the moving configuration and the relative orientation of the lines d and b. Once θ is known, the resulting configuration is easily computed. As before, a suitable nonlinear system of equations is formed and analyzed with Gröbner bases.

5 User Interaction

In general, a well-constrained geometric constraint problem has an exponential number of solutions. For example, consider drawing n points, along with $2n - 3$ distance constraints between them, and assume that the distance constraints are such that we can place the points serially, each time determining the next point by two distances from two known points. In general, each new point can be placed in two different locations: Let p_0 and p_1 be known points from which the

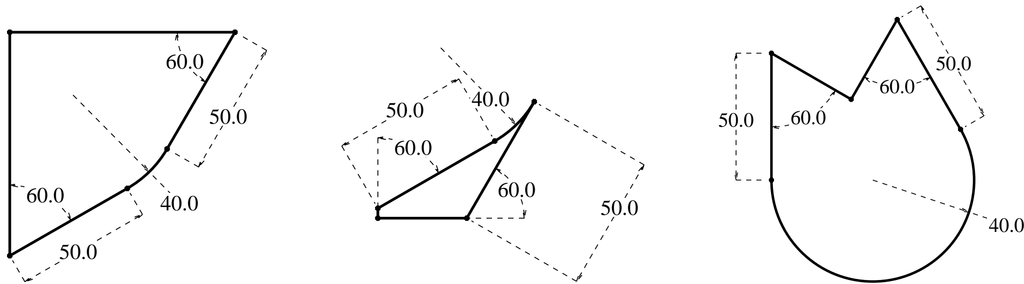

Figure 14: Several Structurally Distinct Solutions of the Same Constraint Problem

new point q is to be put at distance d_0 and d_1, respectively. Draw two circles, one about p_0 with radius d_0, the other about p_1, with radius d_1. The intersection of the two circles are the possible locations of q. For n points, therefore, we could have up to 2^{n-2} solutions. Which of these solutions is the intended one would depend on the application that created the constraint problem in the first place. We discuss how one might select the "right" solution. We call this the *root identification problem*, because on a technical level it corresponds to selecting one among a number of different roots of a system of nonlinear algebraic equations.

Although some solutions of a well constrained problem are merely symmetric arrangements of the same shape, others may differ structurally a great deal. Figure 14 shows several possibilities to illustrate the possible range. But an application will usually require one specific solution. To identify the intended solution is not always a trivial undertaking. Moreover, the wide range of possible solutions has severe consequences on the problem of communicating a generic design based on well-constrained sketches. Since a sketch with a constraint schema would not necessarily identify which solution is the intended one, more needs to be communicated.

In this section, we consider three approaches: selectively moving geometric elements, adding more constraints to narrow down the number of possible solutions, and, finally, a dialogue with the constraint solver that identifies interactively the intended solution. These are approaches that have to contend with some difficult technical problems. We also consider the possibility of structuring the constraint problem hierarchically. Doing so would increase knowledge of the design intent, and would diminish some of the more obvious technical problems.

5.1 Moving Selected Geometric Elements

All constraint solvers known to us adopt a set of rules by which to select the solution that is ultimately presented to the user. Whether stated explicitly, as we will later, or incorporated implicitly into the code of the solver, these rules

ultimately infer which solution would be meant by observing topological and/or coordinate relationships of the initial sketch with which the user specified the constraint problem. When the solution is presented graphically to the user, it seems natural that the user, again graphically, select certain geometric elements of the final sketch that are considered misplaced. The user could then show the solver where the selected element(s) should be placed in relation to other elements by moving them with the mouse.

This very simple idea ultimately may be effective, but there are a number of conceptual difficulties that need to be overcome. For example, picking a geometric element is ambiguous. Because of the recursive nature of the solver, picking could refer to the individual element, or to the cluster or super clusters of which it became part. More importantly, the required restructuring might entail more complex operations than merely moving a single group of geometric elements. Furthermore, since the length of segments and arcs often implicitly depends on the final placement, it is not clear whether the user can reasonably be expected to understand the effect of moving geometries. Clearly more research would be needed to fashion this concept into a useful method.

5.2 Adding More Constraints

Consider once more the problem of placing n points with prescribed distances. We could narrow down which solution is meant in one of two ways: We may add domain knowledge from the application, or we may give additional geometric constraints that actually overconstrain the problem. Unfortunately, both ideas result in NP-complete problems.

For instance, assume that the set of points is the set of vertices of a polygonal cross section. In that case, application-specific information might require that the resulting cross section is a simple polygon; that is, it should form a polygon that is not self-intersecting. This may be communicated by giving, in addition, a cyclical ordering of the points; i.e., the sequence of vertices of the cross section. This very simple additional requirement makes the problem NP-complete:

> **Theorem (Capoyleas)**
> Suppose there are n points in the plane, constrained by $2n-3$ point-to-point distances, and a cyclical ordering specifying how to connect the points to obtain a polygon. The problem of identifying a solution for which the resulting polygon is simple, i.e., is not self-intersecting, is NP-complete.

Consequently, there is little hope for adding domain-specific knowledge about the application with the expectation of obtaining an efficient constraint solver that finds the intended solution in all cases.

Instead of adding application-specific rules, for instance to derive simple polygons, we could add more geometric constraints. For example, consider

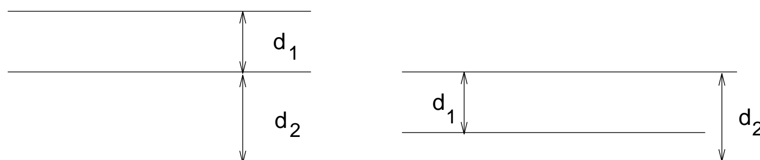

Figure 15: Two Solutions for Three Parallel Lines

specifying three parallel lines along with distances between two pairs of them. As shown in Figure 15, there are two distinct solutions of this well-constrained problem. By adding a required distance between the third pair of parallel lines we can eliminate one or the other case, and make the solution unique.

Overconstrained geometric problems have been carefully avoided by the field because the set of constraints might be contradictory. However, blue prints are usually overdimensioned, although not for reasons of eliminating unwanted solutions, but for limiting errors through redundancy. Again, it is unfortunate that even for the simple case of placing parallel lines the overconstrained problem is NP-complete [9].

Since adding constraints even in such simple situations results in NP-complete problems, it seems to us that the attractive idea of adding more constraints to narrow the range of possible solutions will not work very well in practice. It is plausible that a heuristic approach succeeds in solving this problem in a range of cases that are of practical interest, but always with the possibility that for specific instances the solver would bog down. Again, further research is needed to better understand the potentialities of the approach.

5.3 Dialogue with the Solver

The considerations above seem to suggest that no automatic approach to root identification will succeed in delivering an efficient constraint solver that gets the intended solution every time. Consequently, we feel that a promising alternative is to devise a few simple heuristics that succeed in many cases and are easy to understand. Beyond that, we rely on interaction with the user in those cases in which the heuristics fail to deliver an acceptable solution. Note that placement rules are used very widely, but are rarely discussed.

5.3.1 Placement Heuristics

All solvers known to us derive from the initial geometric sketch information that is used to select a specific solution. This is reasonable, since one can expect that a sketch is at least topologically accurate, so that observing on which side of an oriented line a specific point lies in the sketch is often reliably indicating where it should be in the final solution. However, when generic designs are archived and and later edited, one should no longer expect such

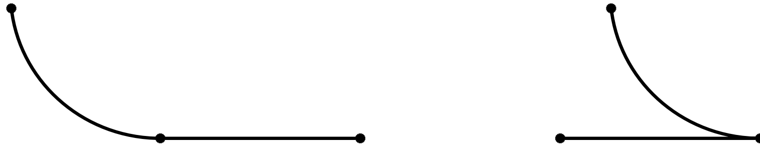

Figure 16: The Two Types of Tangency between an Arc and a Segment

simple correspondences between the sketch and the ultimate solution, because as dimension values change, so may the side of a line on which a point is situated.

In our system, we use very few but highly effective rules. Where the rules fail, we rely on user interaction to amend them as the situation might require. Note that our rules are fully supported by the Erep approach in that the different situations can be characterized and recorded faithfully.

Three Points: Consider placing three points, p_1, p_2 and p_3, relative to each other. The points have been drawn in the initial sketch in some position, and therefore have an order that is determined as follows. Determine where p_3 lies with respect to the line $\overline{(p_1, p_2)}$ oriented from p_1 to p_2. If p_3 is on the line, then determine whether it lies between p_1 and p_2, preceding p_1 or following p_2. The solver will preserve this orientation if possible.

Two Lines and One Point: When placing a point relative to two lines, one of four possible locations is selected based on the quadrant of the oriented lines in which the point lies in the original sketch. Note that the line orientation permits an unambiguous specification of the angle between the lines.

One Line and Two Points: The line is oriented, and the points, p_1 and p_2, are kept on the same side(s) of the line as they were in the original sketch. Furthermore, we preserve the orientation of the vector $\overline{p_1, p_2}$ with respect to the line orientation by preserving the sign of the inner product with the line tangent vector.

Tangent Arc: An arc tangent to two line segments will be centered such that the arc subtended preserves the type of tangency. The two types of tangency are illustrated in Figure 16. Moreover, the center will be placed such that the smaller of the two arcs possible is chosen, ties broken by placing the center on the same side of the two segments as in the input sketch. As specific *degeneracy heuristics*, an arc of length $0°$ is suppressed.

All rules except the tangency rule are mutually exclusive. They are therefore applicable without interference. The tangency rule could contradict the other rules, because dimensioned arcs and circles are determined by placing the center. In such cases, the tangency rule takes precedence. In our experiments with these rules, we found that most situations are solved as the user would expect. The rules are easy to implement, and are easy to understand for the user.

5.3.2 Selecting Alternative Solutions

A useful paradigm for user-solver interaction has to be intuitive and must account for the fact that most application users will not (and should not) be intimately knowledgeable about the technical workings of the solver. So, we need a simple but effective communication paradigm by which the user can interact with the solver and direct it to a different solution, or even browse through a subset of solutions in case the one that was found is not "right."

Conceptually, all possible solutions of a constraint problem can be arranged in a tree whose leaves are the different solutions, and whose internal nodes correspond to stages in the placement of elements or clusters. The different branches from a particular node are the different choices for placing the element or cluster. The tree has depth proportional to the number of elements and clusters. Browsing through all possible solutions would be exponential in the number of elements and would be inappropriate, but stepping from one solution to another one is proportional to the tree depth only.

We have added to our solver an incremental mode in which the user can browse through the construction tree and be visually informed which elements have been placed at a particular moment. With a button, the user steps forward or backwards in the construction sequence, thus traversing the tree path backwards, towards the root, or forward, towards a leaf. At each level, the geometric element(s) placed at that point are highlighted, and a panel displays the number of possible positions. The user can then select which one of the possible choices should be used.

For example, consider the constraint example of Figure 2. The role of the arc is clearly to round the corner that would be formed otherwise by the adjacent segments. When drawn as indicated in the figure, and with angle values larger than 45°, the solver finds the leftmost solution in Figure 14. However, when the angles are changed subsequently to 30°, the solver heuristics will select the solution shown in Figure 17, because the center of the arc remains on the same side of the adjacent segments. The user now relocates the center by changing the placement of Ar_{10} with respect to Sg_7 and Sg_9. By pressing the `level` buttons, the user returns to level 7. Here, Ar_{10} and Sg_7 are highlighted. The user now changes the solution by pressing the `soln.` buttons. This changes the arc center with respect to Sg_7 only. Continuing with the `level` buttons, on level 4, Ar_{10} and Sg_9 are highlighted. Again, a different solution is selected on that level, changing the arc with respect to Sg_9. Now the solver will construct the solution shown in Figure 18. We have found this simple interaction technique highly useful in exploring alternative solutions, and most users become effective in directing the solution process in a very short time.

In the Erep specifications of the interaction process, the solver is instructed serially to perform a back-up or to seek the next way to place an element or a cluster. This convention excludes solvers that find a solution by a numerical,

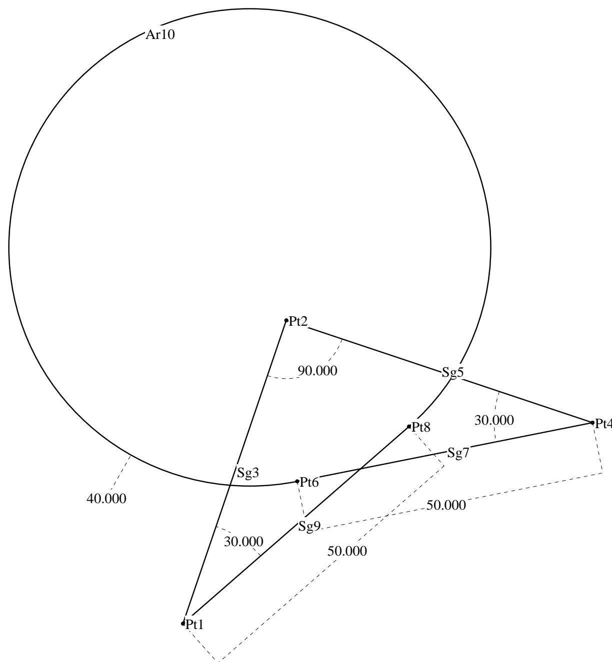

Figure 17: Default Solution After Changing Angles

iterative computation which places all geometric elements at once, unless the iteration is based on homotopy continuation techniques that can find all solutions of a nonlinear system of equations numerically.

Note that different solvers may cluster geometric elements differently and place the elements and clusters in a different sequence. Therefore, the same interaction sequence with the solver would have different effects with different solving algorithms. This cannot be avoided: To arrange the tree of solutions in canonical order, we either prescribe a canonical sequence *a-priori* in which the geometric elements have to be computed, or else we compute a canonical basis for the ideal generated by the constraint equations that describe the geometric problem, and then enumerate the associated variety in a canonical way; e.g., [4]. In the first case, we would prescribe the solver algorithm to belong to a certain family. In the second case, the ideal basis computation is equivalent to solving the constraint problem and thus constitutes committing to a canonical solver.[2] Both ways compromise devising a neutral format of archiving. Consequently, we can neutrally archive a constraint problem (solved or unsolved), but not the manner in which to solve or seek an alternative solution. This is an intrinsic problem when solving geometric constraints.

[2] Because such a canonical solver would be completely general, it could not be very fast in many situations, since the efficiency of constraint solvers rests on restricting the generality of the solver.

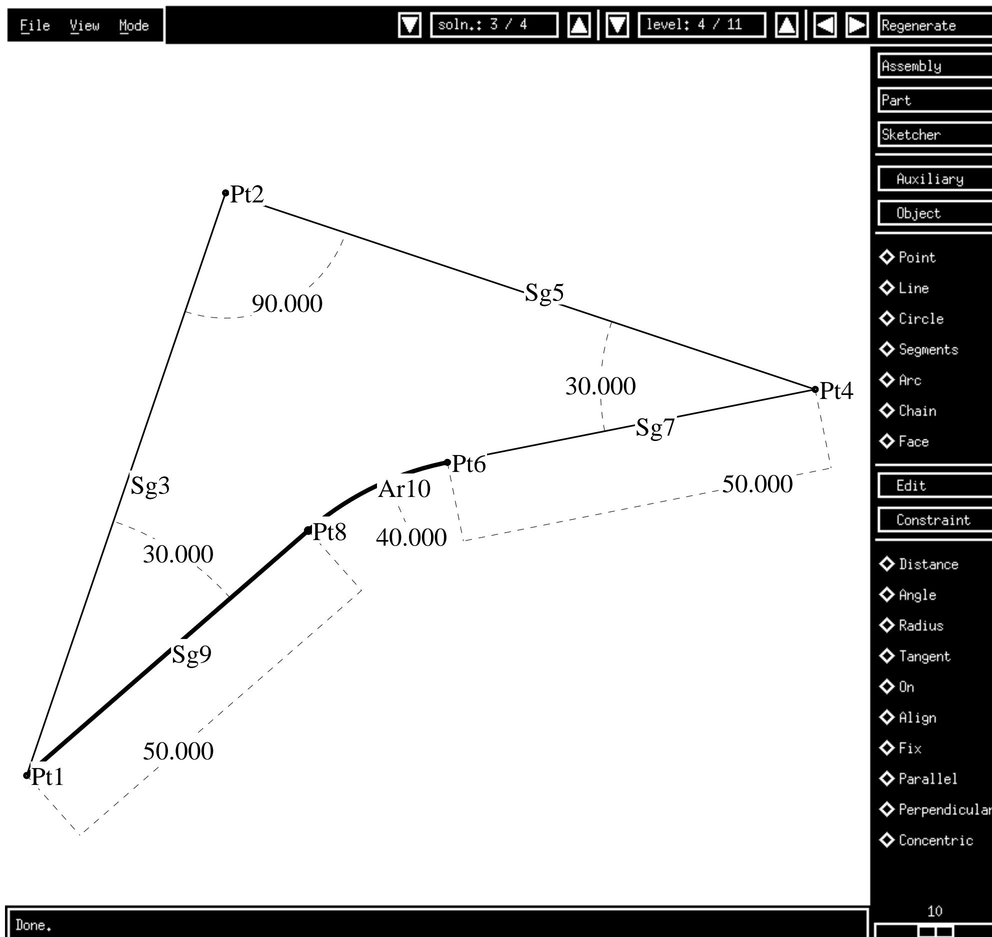

Figure 18: Interactively Changing Solutions: Elements highlighted are placed with respect to each other at this level in the solution tree.

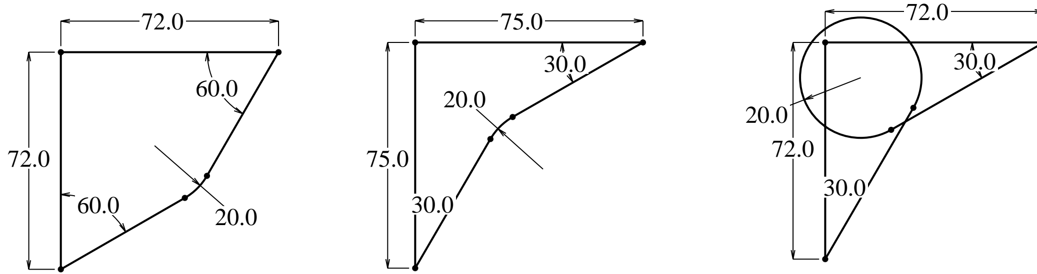

Figure 19: A constraint problem, left, and two solutions after changing angles.

5.4 Design Paradigm Approach

Consider solving the constraint problem of Figure 19 (left). The role of the arc is clearly to round the adjacent segments, and thus it is most likely that the solution shown in Figure 19 in the middle is the one the user meant rather than the one on the right, when changing the angles to 30°. The solver would be unaware of the intended meaning of the arc, and thus needs a technical heuristic, such as the tangent arc rule, to avoid the solution on the right. It would be much simpler if the user would sketch in such a way that the design intent of the arc is evident.

The difficulty for the geometric constraint solver is that sketches are usually *flat*; i.e., the geometric elements are not grouped into "features." It would be better to make sketches hierarchically: First, a basic dimension-driven sketch would be given. Then, subsequent steps, also dimension-driven, would modify the basic sketch and add complexity. In our example, the basic sketch could be a quadrilateral. There would be one subsequent modification adding a two-dimensional *round* with a required radius. This is analogous to feature-based design as implemented in current CAD systems.

The hierarchical approach to sketching has other important benefits. Since the sketch is structured, later modifications can be driven from constraints used in earlier steps, so that simple functional dependencies and relations between dimension variables of previously defined sketch features can be defined and implemented with trivial extensions of our basic constraint solver.

6 Summary and Future Work

Research on constraint solving should develop natural paradigms for narrowing down the number of possible solutions of a well-constrained geometric problem and devising solver interaction paradigms that allow the user to correct solutions that were not intended. With increasing penetration of constraint-based design interfaces, this problem is becoming increasingly more pressing.

Which solution is the intended one is also crucial for archiving designs in a

neutral format. So far, neutral archiving has been restricted to detailed design without a formal record of design intent, constraint schema, editing handles, and so on. When editable design is archived, it requires a proprietary format native to the particular CAD system, and is typically a record of the internal data structures of the CAD system. In [13] we have presented alternatives. Current trends in data exchange standards indicate a growing interest in archiving constraint-based designs in which this additional information has been formalized without commitment to a particular CAD system.

In constraint-based, feature-based design, it is common to have available a variational constraint solver for 2D constraint problems, but not for 3D geometric constraints. This is particularly apparent in the *persistent id problem* discussed in [12]. A well-conceived 3D constraint solver conceivably can avoid these problems and assist in devising graphical techniques for generic design.

In manufacturing applications one is interested in functional relationships between dimension variables, because such relationships can express design intent very flexibly. Some parametric relationships can be implemented easily by structuring the sketcher as advocated in Section 5.4. Moreover, simple functional relationships are the content of certain geometry theorems, such as the theorems of proportionality, and many other classical results. Such theorems can be added to the constraint solver in a manner analogous to the extensions we have discussed before. But in general, functional relationships between dimension variables necessitate additional mathematical techniques. Geometric theorem proving has developed many general techniques that are applicable, but suitable restrictions are still needed to achieve higher solver speeds.

Geometric coverage refers to the range of shapes the constraint solver understands. In this work, we have restricted the geometric coverage to points. lines and circles. Conic sections would be easy to add, as would be splines such as Bézier curves, when translating the constraints to equivalent ones on control points. There is a rich repertoire of literature in CAGD that provides convenient tools for doing so. Yet it is not clear whether control point manipulations are a natural tool for expressing constraints. We miss studies that analyze how to design with splines from an application's point of view.

Even with the resticted geometric coverage discussed here, some theoretical problems remain open. Although no precise analysis has been made, neither Owen's nor our constraint solving algorithm seems to run in worst case time linear in the number of graph edges. We conjecture that both algorithms run in quadratic time due to repeated traversals over regions of the graph. It would be worthwhile to analyze the worst case running times of these algorithms precisely, and study how to improve them. It is also worthwhile to consider how to minimize the arithmetic operations involved in the construction steps, and to analyze construction sequences for numerical stability.

Acknowledgement

We had several insightful discussions with John Owen from D-Cubed, Ltd.

References

[1] B. Aldefeld. Variation of geometries based on a geometric-reasoning method. *Computer Aided Design*, 20(3):117–126, April 1988.

[2] P. Borras, D. Clement, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. Centaur: the system. Technical Report Rapports de Recherche 777, INRIA, 1987.

[3] B. Bruderlin. Constructing Three-Dimensional Geometric Objects Defined by Constraints. In *Workshop on Interactive 3D Graphics*, pages 111–129. ACM, October 23-24 1986.

[4] B. Buchberger. Gröbner Bases: An Algorithmic Method in Polynomial Ideal Theory. In N. K. Bose, editor, *Multidimensional Systems Theory*, pages 184–232. D. Reidel Publishing Co., 1985.

[5] B. Buchberger, G. Collins, and B. Kutzler. Algebraic methods for geometric reasoning. *Annual Reviews in Computer Science*, 3:85–120, 1988.

[6] J. Cai. A language for semantic analysis. Technical Report 635, New York University, Dept. of Comp. Science, 1993.

[7] J. Cai and R. Paige. Towards increased productivity of algorithm implementation. ACM SIGSOFT, 1993.

[8] C.-S. Chou. *Mechanical Theorem Proving.* D. Reidel Publishing, Dordrecht, 1987.

[9] I. Fudos. Editable representations for 2d geometric design. Master's thesis, Purdue University, Dept. of Comp. Sci., 1993.

[10] I. Fudos and C. M. Hoffmann. Correctness proof of a geometric constraint solver. Technical Report 93-076, Purdue University, Computer Science, 1993.

[11] C. M. Hoffmann. *Geometric and Solid Modeling.* Morgan Kaufmann, San Mateo, Cal., 1989.

[12] C. M. Hoffmann. On the semantics of generative geometry representations. In *Proc. 19th ASME Design Automation Conference*, pages 411–420, 1993. Vol. 2.

[13] C. M. Hoffmann and R. Juan. Erep, a editable, high-level representation for geometric design and analysis. In P. Wilson, M. Wozny, and M. Pratt, editors, *Geometric and Product Modeling*, pages 129–164. North Holland, 1993.

[14] D. Kapur. A refutational approach to geometry theorem proving. In D. Kapur and J. Mundy, editors, *Geometric Reasoning*, pages 61–93. M.I.T. Press, 1989.

[15] D. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2:127–145, 1968.

[16] D. Knuth and P. Bendix. Simple word problems in universal algebras. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297. Pergammon Press, Oxford, 1970.

[17] K. Kondo. PIGMOD: parametric and interactive geometric modeller for mechanical design. *Computer Aided Design*, 22(10):633–644, December 1990.

[18] K. Kondo. Algebraic method for manipulation of dimensional relationships in geometric models. *Computer Aided Design*, 24(3):141–147, March 1992.

[19] G. Kramer. *Solving Geometric Constraint Systems*. MIT Press, 1992.

[20] W. Leler. *Constraint Programming Languages: Their Specification and Generation*. Addison Wesley, 1988.

[21] R. Light and D. Gossard. Modification of geometric models through variational geometry. *Computer Aided Design*, 14:209–214, July 1982.

[22] J. Owen. Algebraic solution for geometry from dimensional constraints. In *ACM Symp. Found. of Solid Modeling*, pages 397–407, Austin, Tex, 1991.

[23] R. Paige. Apts external specification manual. internal documentation, 1993.

[24] Reasoning Systems. *Refine User's Guide*, 1990. Version 3.0.

[25] T. Reps and T. Teitelbaum. *The Synthsizer Generator*. Springer Verlag, 1988.

[26] A. Requicha. Dimensionining and tolerancing. Technical report, Production Automation Project, University of Rochester, May 1977. PADL TM-19.

[27] K. Snyder. The SETL2 programming language. Technical report, New York University, Computer Science, Courant Institute, 1990.

[28] W. Sohrt. Interaction with Constraints in three-dimensional Modeling. Master's thesis, Dept of Computer Science, The University of Utah, March 1991.

[29] L. Solano and P. Brunet. A system for constructive constraint-based modeling. In B. Falcidieno and T. Kunii, editors, *Modeling in Computer Graphics*, pages 61–84. Springer Verlag, 1993.

[30] G. Sunde. Specification of shape by dimensions and other geometric constraints. In M. J. Wozny, H. W. McLaughlin, and J. L. Encarnacao, editors, *Geometric Modeling for CAD Applications*, pages 199–213. North Holland, IFIP, 1988.

[31] I. Sutherland. Sketchpad, a man-machine graphical communication system. In *Proc. of the spring Joint Comp. Conference*, pages 329–345. IFIPS, 1963.

[32] P. Todd. A k-tree generalization that characterizes consistency of dimensioned engineering drawings. *SIAM J. DISC. MATH.*, 2(2):255–261, 1989.

[33] A. Verroust, F. Schonek, and D. Roller. Rule-oriented method for parameterized computer-aided design. *Computer Aided Design*, 24(3):531–540, October 1992.

[34] Wu Wen-Tsün. Basic principles of mechanical theorem proving in geometries. *J. of Systems Sciences and Mathematical Sciences*, 4:207–235, 1986.

[35] Y. Yamaguchi and F. Kimura. A constraint modeling system for variational geometry. In M. J. Wozny, J. U. Turner, and K. Preiss, editors, *Geometric Modeling for Product Engineering*, pages 221–233. Elsevier North Holland, 1990.