An Interactive N-Dimensional Constraint System

Ching-yao Hsu
Beat Bruderlin

UUCS-94-036

Department of Computer Science
University of Utah
Salt Lake City, UT 84112 USA

December 8, 1994

Abstract

In this paper, we present a graph-based approach to geometric constraint solving. Geometric primitives (points, lines, circles, planes, etc.) possess intrinsic degrees of freedom in their embedding space. Constraints reduce the degrees of freedom of a set of objects. A constraint graph is created with objects as the nodes, and the constraints as the arcs. A graph algorithm transforms the undirected constraint graph into a directed acyclic dependency graph which can be directly used to derive a sequence of construction operations as a symbolic solution to the constraint problem. The approach has been generalized to an n-dimensional space, which, among other things, allows for a uniform handling of 2-D and 3-D constraint problems or algebraic constraints between scalar dimension. Solutions of arbitrary dimensions can be interpreted as approaches to over- and under- constrained problems. In this paper, we present the theoretical background of the approach, and report the results of it's application within an interactive modeling system.

An Interactive N-Dimensional Constraint System

Ching-yao Hsu, Beat Brüderlin
Department of Computer Science
University of Utah
Salt Lake City, Utah 84112

Abstract

In this paper, we present a graph-based approach to geometric constraint solving. Geometric primitives (points, lines, circles, planes, etc.) possess intrinsic degrees of freedom in their embedding space. Constraints reduce the degrees of freedom of a set of objects. A constraint graph is created with objects as the nodes, and the constraints as the arcs. A graph algorithm transforms the undirected constraint graph into a directed acyclic dependency graph which can be directly used to derive a sequence of construction operations as a symbolic solution to the constraint problem. The approach has been generalized to an n-dimensional space, which, among other things, allows for a uniform handling of 2-D and 3-D constraint problems or algebraic constraints between scalar dimension. Solutions of arbitrary dimensions can be interpreted as approaches to over- and under- constrained problems. In this paper, we present the theoretical background of the approach, and report the results of it's application within an interactive modeling system.

1 Introduction

Conventional modeling systems did not support the free dimensioning of geometric objects by means of constraints, but require users to construct them by a sequence of geometric operations. Mechanical parts designed by such a CAD system are represented as fixed geometry; the geometric design is completely separated from other design criteria. It is often difficult for a user to determine the exact coordinates of the objects in the beginning or to add information under a different view, later on. Changing a part may inadvertently violate previous design decisions. Most computer aided design systems, nowadays, allow one to define geometric constructions by means of parameters. The value of the design parameters can be determined later, and a dependency propagation mechanism automatically propagates the new values to all directly and indirectly dependent parts of the object. Although this increases the flexibility of CAD based design significantly, great care has to be taken to define the geometric operations in the right order, which puts an undue burden on the designer. Surveys of these works can be found, for instance, in [26, 19, 27].

Geometric constraints have shown to be useful for interactive geometric design. The idea is to specify shape by constraints such as distances, angles, etc. and use a constraint solver to derive the shape from such a specification. A clear drawback of a constraint based approach is that it is extremely difficult and not at all intuitive for a designer to come up with a complete and consistent set of constraints. Often we encounter over and under specified parts simultaneously that are hard to resolve in a specification. Also constraint solving is a very difficult problem, even if the specification is consistent. Following is an brief survey on different kinds of constraint solving techniques in the literature.

1.1 Constraint Solving Methods

Constraint propagation is one of the basic mechanisms used in early constraint based system for the derivation of solutions that satisfy the given constraints. Here the system of variables and constraints are represented as an undirected graph. The nodes of the graph represent variables or constants, and the edges represent equations relating the variables and constants. The solving of constraints is done by finding an order of evaluation to satisfy all the equations from the constants progressively. Propagation methods are described, for instance, in [3, 8, 9, 11, 21, 18, 30]. The weakness of the propagation approach is that it can not handle cyclic constraint situations, and hence it is usually coupled with numerical methods as described below.

Most constraint based systems use numerical techniques (e.g., relaxation, Newton-Raphson iteration) which can theoretically solve problems even if they don't have a closed form algebraic or geometric solution. In this approach, constraints are translated into a system of algebraic equations and then solved using iterative methods. Numerical approaches are described in [12, 13, 22, 23, 28]. While they are quite powerful and general, numerical techniques have convergence problems that make them very unpredictable.

Because of its generality, a lot of systems switch to numerical method when their basic mechanisms failed. Early systems such as Sketchpad, ThingLab and Magritte used relaxation as an alternative to their propagation methods.

Lately, another kind of constraint solvers is emerging, which satisfies the constraints using a sequence of construction steps and solves problems solvable by ruler and compass construction. It can be viewed as an extension of the constraint propagation paradigm into higher dimensions. The basic principle behind constraint propagation is that an object can be evaluated when enough information about it is available. These methods differ in the way the order of

constraint type	arity	valency
distance	2	1
slope	2	1
vector	2	2
angle	3	1
midpoint	3	2

Table 1: Constraint types and their valencies.

evaluation is determined, and can be roughly divided into two categories, the rule-based approach and the graph-based approach. Work belonging to this group can be found in [2, 4, 10, 14, 5, 6, 7, 20, 25, 31, 24].

1.2 An Overview

It is our objective to develop an approach for evaluating the degrees of freedom of under-constrained networks of constraints so that user can draft in a less restricted way in the early design stage. We will show that with this approach, users are not forced to specify shapes completely by constraints but can add constraint definitions incrementally, and manipulate the geometric models within their degrees of freedom.

The graph-based approach developed in this paper is based on the law of conservation (see section 3.1). Section 2 introduces the basic definitions and concepts. Section 3 develops the basic algorithm for the degree of freedom analysis of a constraint network; it returns an intermediate representation, the dependency graph. Section 4 is concerned with some details of the basic algorithm and properties of the dependency graphs. Section 5 suggests ways to evaluate the dependency graph by a sequence of geometric construction operations. Section 6 presents several extensions to the basic algorithm to represent algebraic equations, and congurence relations. Section 7 deals with a 3-D application of the algorithm.

2 Basic Definitions and Concepts

2.1 Geometric Model

A *parametric geometric model* is defined as a collection $O = \{ o_1, o_2, \ldots, o_k \}$ of geometric objects and a set $C = \{ c_1, c_2, \ldots, c_m \}$ of geometric constraints among the members of O.

Geometric objects such as points, lines, and circles *own* degrees of freedom, which allow them to vary in size, shape, orientation, or position. The set $DOF = \{ dof_1, dof_2, \ldots, dof_k \}$ of degrees of freedom owned by objects in O completely determine the *set of states* of the geometric model.

Geometric constraints such as those defined for 2-D points in Table 1 define an n-ary geometric relation among a set of n objects, $c_i = c_i(o_{i_1}, o_{i_2}, \ldots, o_{i_n}, \lambda)$, where λ is the parameter of the constraint. Depending on the constraint type, the parameter may be a single scalar value or a vector. A constraint reduces the degrees of freedom from the set of objects by a certain number. This number will be called the *valency* of the constraint, as defined in [2]. In general, a valency of a constraint c_i is a positive number which is less than or equal to the sum of the degrees of freedom owned by the n objects.

$$0 \leq valency(c_i) \leq \sum_{j=1}^{n} DOF(o_{i_j}) \qquad (1)$$

where $DOF(o_{i_j})$ refers to the degrees of freedom of the object o_{i_j}, and $valency(c_i)$ denotes the valency of the constraint c_i. If the valency is equal to the total number of the degrees of freedom owned by the n objects, we call the constraint a *full-valency constraint*. A full-valency constraint completely determines the state of the objects constrained.

We also define separately a special class of constraints, *local constraints*, which are unary constraints *consuming* all or part of the degrees of freedom owned by an object. For example, a constant x-coordinate constraint on a 2-D point fixes its x-coordinate in space; therefore we say that one degree of freedom owned by the point is consumed by that constraint. If the local constraint consumes all the degrees of freedom, we call it a *full local constraint*.

2.2 Constraint Network

In the following sections, we will use constraint networks as the graph representation of a geometric model.

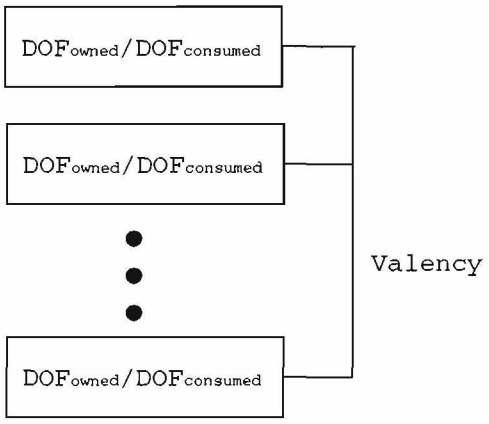
A *constraint network* is an undirected graph which consists of a finite set of nodes and a finite set of arcs. A *node* in the network represents an object and is depicted as a rectangle box with two numbers separated by a slash. The first number, DOF_{owned} denotes the degrees of freedom owned by the unconstrained object; the second number, $DOF_{consumed}$ denotes the degrees of freedom consumed by the local constraints, if there are any. An n-ary relation is represented by an undirected *arc* with fan-out equal to n. We label the arc with the valency of the constraint on the side as shown in Figure 1. In the following sections, we will use object and node, as well we constraint and arc interchangeably.

If the constraint specification is consistent and non-redundant, a constraint network is said to be *fully constrained*, if the total number of degrees of freedom is equal to that of the valencies. On the other hand, to constrain n points relative to each other in 2-D space, for example, only $2n$ - 3 distance or angle constraints are required. The solution will be a rigid body with three remaining degrees of freedom. If we ignore rigid body transformations for the moment, we can define well-constrained, under-constrained, and over-constrained constraint network informally as follows: A constraint network is *well-constrained*, if the number of states of the constraint network is finite. If some of the objects lie in a continuum, and hence there are infinite number of states, then the constraint network is said to be *under-constrained*. Finally, the constraint network is *over-constrained* if there is no valid state to satisfy the constraint network. Note that a constraint network can be partly under-constrained and partly over-constrained at the same time.

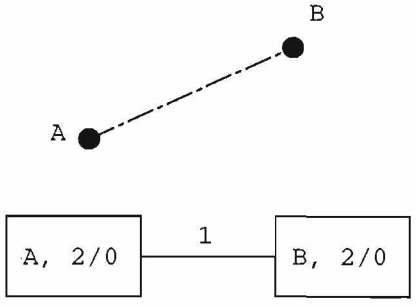
For an under-constrained network, there will be many degrees of freedom *stored* in it. We will be able to change the state of the constraint network by manipulating objects within all or a subset of the degrees of freedom stored. Different portions of the constraint network will be involved if different subsets of the degrees of freedom are used.

2.3 The Degrees of Freedom Stored

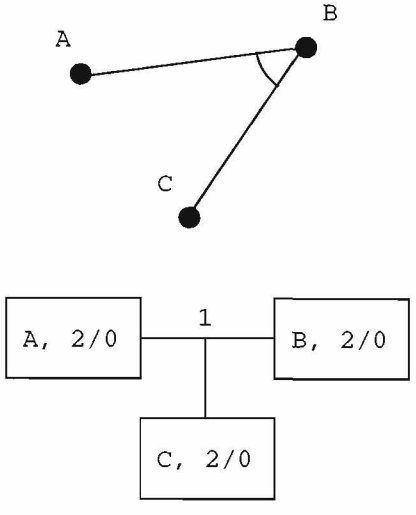
Before discussing how a portion of a constraint network can be manipulated, the definition of the degrees of freedom



(a) The representation



(b) A distance constraint



(c) An angle constraint

Figure 1: The nodes and arcs of the constraint networks.

stored in a connected component of a constraint network is introduced.

The notion of a connected component of a constraint network is defined with respect to the degrees of freedom stored. When a node is fully constrained, it is fixed in space and can no longer be manipulated. We refer to these nodes as the *dead nodes*. A dead node will potentially make the constraint network disconnected in terms of degrees of freedom. There are two situations when a node becomes fully constrained:

1. When there is a full local constraint.
2. When it is fully constrained by fully constrained neighbors.

We define two objects, o_1 and o_n, to be *connected* if

- there exists a sequence of objects $o_1, o_2, \ldots, o_n$ such that there is at least one constraint defined between o_i and o_{i+1} for $1 \leq i < n$, and
- none of the o_i's are dead nodes, for $1 \leq i < n$.

A connected component of a constraint network is defined to be a maximal connected induced subgraph as defined in traditional graph theory [1] except that the notion of connection is sharpened as above.

If G is a connected component of a constraint network, the degrees of freedom stored in G can be calculated as

$$DOF(G) = \sum_{o \, in \, G} DOF_{owned}(o) - \sum_{o \, in \, G} DOF_{consumed}(o) - \sum_{c \, in \, G} valency(c) \quad (2)$$

3 Degrees-of-Freedom Analysis

The goal of the degrees of freedom analysis is to extract from a constraint network a connected portion which possesses a specified degrees of freedom. We can then manipulate the portion of the constraint network through the degrees of freedom acquired.

The algorithm presented below, is based on the law of conservation[1]. We will first establish the balance equation for any quantity which observes the law of conservation. We will then derive the balance equations for the degrees of freedom of the nodes and arcs of the constraint network. At the end, we will present the degrees-of-freedom analysis algorithm which will generate a dependency graph as the result.

3.1 The Balance Equation

Some quantity χ, with respect to a system, can either enter the system or leave the system. It can also be generated or destroyed within the system (as shown in figure 2). We can express the conservation law for χ by the so-called balance equation:

$$\text{output of } \chi = \text{input of } \chi + \chi \text{ generated} - \chi \text{ destroyed} - \chi \text{ accumulated} \quad (3)$$

If the system does not change with time, or in other words, the system is operated under *steady-state conditions*, equation 3 can be simplified with respect to the accumulation

[1] Please refer to [15, 16] for a similar algorithm developed from a 2-D geometric point of view.

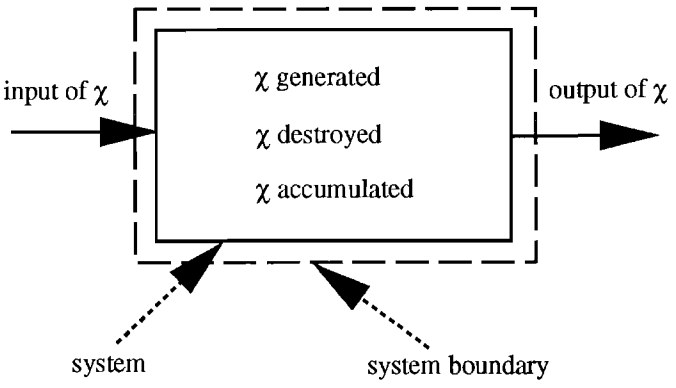


Figure 2: A system with input and output.

term. Since an accumulation means that the amount of the quantity χ in the system is increasing or, in the opposite sense, decreasing, by the definition of steady-state the accumulation term must be zero. Therefore, the balance equation 3 can be shortened to:

$$\text{output of } \chi = \text{input of } \chi + \chi \text{ generated} - \chi \text{ destroyed} \quad (4)$$

For background information on the balance equation, we refer to text books on fluid flow or heat transfer.

3.2 The Dependency Graph

A *dependency graph* is a directed, acyclic hyper-graph derived from the constraint network. Nodes and arcs in the dependency graph are systems, obeying the law of conservation of degrees of freedom, and therefore we can express a balance equation for degrees of freedom for each of them. A *node* in the dependency graph represents an object and is depicted as a trapezoidal box (figure 3) with the long side representing the input side and the short side representing the output side. There are three rows of numbers in the box. The top row enumerates the individual degrees of freedom leaving the box for the next level up. The bottom row shows the individual degrees of freedom entering the box from the previous level down. The center row is the balance equation for the node.

An *arc* (figure 4) represents a constraint between at least one child node and a single parent node. The arc is labeled with the negative valency of the constraint and the arrow at the top shows the direction of the flow. For an *n*-ary constraint, there will be *n - 1* child nodes. However, the fan-out is always one.

We can write balance equations for both nodes and arcs based on the law of conservation of degrees of freedom. Therefore by equation 4, for a node, we have (see also Figure 3),

$$DOF_{out} = DOF_{in} + DOF_{owned} - DOF_{consumed} \quad (5)$$

$$DOF_{in} = \sum_i DOF_{in}(i)$$

$$DOF_{out} = \sum_j DOF_{out}(j)$$

For an arc (see also Figure 4),

$$DOF_{out} = DOF_{in} - valency \quad (6)$$

$$DOF_{in} = \sum_k DOF_{in}(k)$$

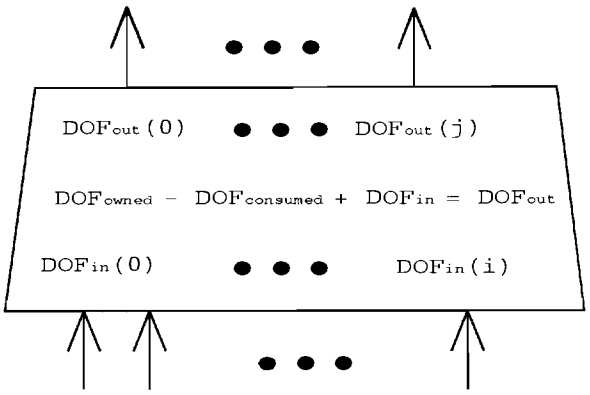


Figure 3: The node of a dependency graph.

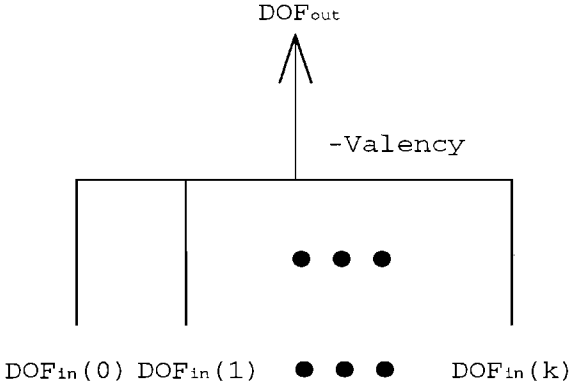


Figure 4: The arc of a dependency graph.

Here, we treat DOF_{owned} as the degrees of freedom generated by the object at constant rate, and $DOF_{consumed}$ as the degrees of freedom destroyed at constant rate within the object. Likewise, the valency is the degrees of freedom destroyed at constant rate within an arc. The incoming degrees of freedom which are zero or negative can be interpreted as the amount by which the degrees of freedom of the node is restricted.

4 An Algorithm for Constructing the Dependency Graph

As stated earlier, our goal is to extract from the constraint network a connected portion that possesses a given degree of freedom. In the algorithm discussed below, the connected portion will be represented as a dependency graph. This is an inverse problem, since we determine how many degrees of freedom we request from the *root node*, i.e. the topmost node. The task of the algorithm is it, to construct the rest of the dependency graph, yielding the desired output. Therefore, a top-down approach seems to be the natural choice. At each node or arc, we know how many degrees of freedom leaving the system, and given the degrees of freedom generated and destroyed in the system, we will be able to calculate the degrees of freedom entering the system. These degrees of freedom, in turn, become the degrees of freedom leaving the systems in the previous level down.

To avoid cyclic dependency (i.e. a node occurs more than once in the path of the directed acyclic graph), we will construct the dependency graph in a breadth-first manner. For more information, please refer to [17].

4.1 The Algorithm

First, we pick an object from the constraint network and request a specified degree of freedom, called $DOF_{requested}$, from it.

We make the object the root node and compute the amount of degrees of freedom that should enter the node giving the amount leaving by rearranging equation 5:

$$DOF_{in}(root) = DOF_{out}(root) - DOF_{owned}(root) + DOF_{consumed}(root) \quad (7)$$

where $DOF_{out}(root) = DOF_{requested}$. The amount DOF_{in} determines the total number of degrees of freedom we need to acquire from the child nodes in order to satisfy the request.

If m constraints, $c_1, c_2, \ldots, c_m$ are attached to this object, we then expand the node by distributing the DOF_{in} among all constraints[2] and creating their corresponding arcs:

$$DOF_{in}(root) = DOF_{out}(c_1) + DOF_{out}(c_2) + \cdots + DOF_{out}(c_m) \quad (8)$$

Notice that if the node expanded is not the root node, we need to exclude from the set of the m constraints any constraints which have already been used elsewhere in the dependency graph.

For arc i, we can calculate the amount of degrees of freedom entering by rearranging equation 6:

$$DOF_{in}(c_i) = DOF_{out}(c_i) - valency(c_i) \quad (9)$$

We then divide $DOF_{in}(c_i)$ provided the number of the fan-in of the arc is greater than one. Let the number of the fan-in be equal to n, and the child nodes are $o_1, o_2, \ldots, o_n$. Then

$$DOF_{in}(c_i) = DOF_{out}(o_1) + DOF_{out}(o_2) + \cdots + DOF_{out}(o_n) \quad (10)$$

where $DOF_{out}(o_j)$ is the degrees of freedom leaving the child node j. It can also be regarded as the request of degrees of freedom to this node.

We will repeat this process down the branches until either of the following conditions is met:

1. There are no more unused constraints, or
2. DOF_{out} of a node is less than or equal to zero.

When one of those conditions is met, we can stop recursion and make the node a *leaf node*. If the second condition is met, we also set $DOF_{consumed}$ equal to DOF_{owned} (i.e. to locally fix the state of the object).

Note that if there are a mix of positive and negative degrees of freedom in the list of DOF_{out} in a node, we first store the negative degrees of freedom away, and set it to zero. After the dependency graph is constructed, we pass the negative degrees of freedom up to its parent, and reset $DOF_{consumed}$ of the parent node.

4.2 An Example

Figure 5 (a) shows a simple geometric model consisting of five points and four distance constraints in 2-D space. Figure 5 (b) shows the corresponding constraint network.

Figure 5 (c) shows one instance of the possible dependency graphs derived by requesting one degree of freedom from point C. In order to satisfy this request, we need to import minus one degrees of freedom from the neighborhood of point C, i.e. point B and point D. We can choose from the following combinations to meet the requirement:

- requesting -1 from point B and 0 from point D,
- requesting 0 from point B and -1 from point D,
- requesting -2 from point B and 1 from point D, etc.

The figure shows the first combination. For point B, the request for degrees of freedom is zero, which signifies a termination condition. Therefore, we set $DOF_{consumed}$ equal to DOF_{owned} for point B, and stop recursion. By applying the algorithm in this manner, we achieve the dependency graph shown.

4.3 The Degrees Of Freedom Requested

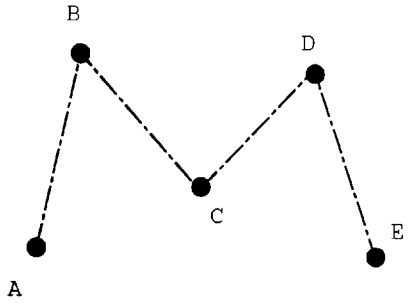
The maximum degrees of freedom we are able to attain from a node in the constraint network will depend on the number of degrees of freedom stored in the connected neighborhood of that node.

We define the *connected neighborhood of an object o* to be the maximal connected induced subgraph which contains o. Suppose G is the connected neighborhood of the object o. The *maximum attainable degree of freedom* of the object o is then

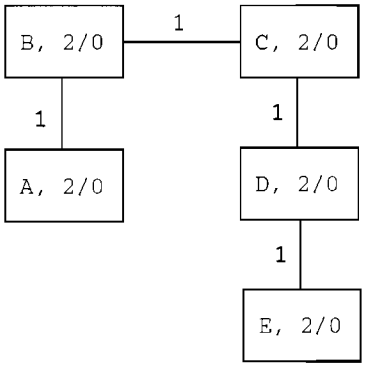
$$DOF_{max}(o) = DOF(G) \quad (11)$$

On the other hand, the lower bound on the degrees of freedom that can be requested is zero. There are two ways of constructing the dependency graph if zero degrees of freedom are requested. First, if the algorithm above is applied, DOF_{out} equal to zero is one of the termination conditions.

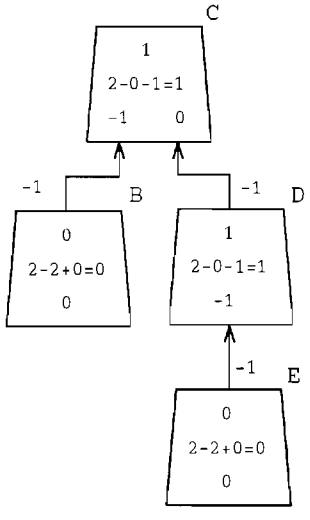
[2] This notion will be made clear in section 4.4.



(a) The model

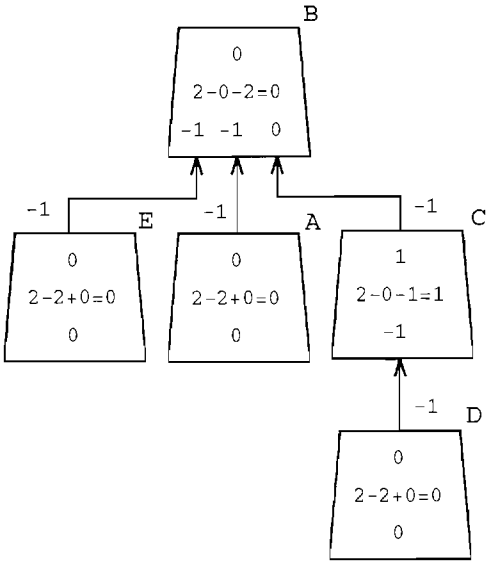


(b) The constraint network

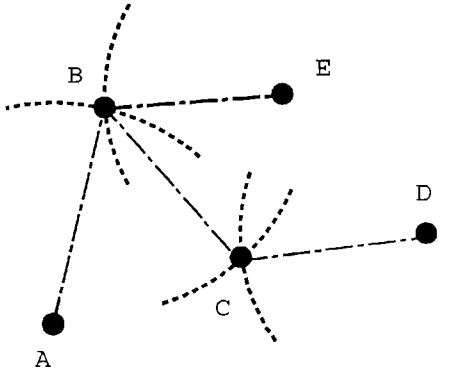


(c) The dependency graph

Figure 5: An instance of the dependency graph.



(A) The dependency graph



(b) The model

Figure 6: Requesting zero degrees of freedom.

Therefore, the dependency graph consists of the node only. Alternatively, we can set DOF_{in} of the root node to be equal to the negative of the net degrees of freedom in that object, and conduct the algorithm as usual from this point on. An example is shown in Figure 6.

Finally, it is, in fact, possible to construct a dependency graph with a negative degree of freedom. However, the dependency graph represents an over-constrained system and is therefore not useful in common applications.

4.4 Search Strategy with Backtracking

The algorithm is non-deterministic in nature because there is, in general, more than one way of dividing the degrees of freedom as illustrated in equation 8 and 10. However, not all of the combinations are valid. In the following, we will introduce a division method that can be used in an exhaustive search. More sophisticated search methods are possible, but require lengthy treatment. Hence we will not discuss them here.

To do the exhaustive search, we basically generate all possible distributions in some order. A partial order is defined such that the most uniform distribution is lowest. A total order is derived from that by refining the partial order by using lexicographic sorting. Since the division is done

blindly, there will be circumstances when a request for a degree of freedom fails. For example, it is a contradiction when a positive degree of freedom is requested from a dead node because the node possesses no degrees of freedom. Backtracking is used to search for alternative dependency graphs under these circumstances.

The described procedure would potentially generate infinitely many dependency graphs. By enforcing the following rules, we limit the set of solutions to be finite:

- The total amount requested may not exceed that stored in the object.

$$DOF_{out} \leq DOF_{owned} - DOF_{consumed} \quad (12)$$

- No positive degrees of freedom are imported.

$$DOF_{in}(i) \leq 0 \text{ for all i} \quad \textbf{(13)}$$

These rules also guarantee that the dependency graphs generated are valid.

5 Evaluating the Symbolic Solution

Once we have a dependency graph, we will be able to evaluate it under various conditions, for instance when manipulating part of the constraint network interactively within the degrees of freedom acquired.

The dependency graphs produced by the exhaustive search will have some *extra* degrees of freedom (the difference between the degrees of freedom of the dependency graph and the one requested). These extra degrees of freedom are easily identified in the dependency graphs since they are all stored in the leaf nodes. As a consequence, we can manage to offset these extra degrees of freedom acquired easily. For one thing, we can impose temporary local constraints on those leaf nodes. Once we have process all the nodes in this manner, the degrees of freedom of the dependency graph will be equal to that of requested.

For the following discussion, we will assume that the degrees of freedom of the dependency graph is equal to that of requested. These degrees of freedom acquired are at our disposal. For instance, if we use an interactive user interface which supports locator devices, we can use them to interactively specify these degrees of freedom. In particular, if two degrees of freedom are requested from a 2-D point, we can manipulate the coordinates of the point by assigning them the coordinates of the locator device.

After fixing the degrees of freedom acquired, the dependency graph itself becomes a fully constrained system. We can solve for the new state of the dependency graph by using a variety of established methods for solving fully constrained systems.

We will again use the example shown in Figure 5 to describe how different kinds of constraint solvers are used to evaluate the dependency graph.

- Constructive constraint solver:

  In constructive constraint solvers, the constraint network is satisfied using step-by-step constructions. To solve the dependency graph shown in Figure 5 (c), we first pin down the one degree of freedom of point C interactively. Since point B is fixed, point C must lie on the circle centered at point B. In other words, it only takes one parameter to completely determine the position of point C. After that is done, the position of point D can be determined by intersecting the two circles centered at point E and new point C respectively. For more detailed information, please refer to [15, 16]. Generally we try to find solutions as intersections of circles and lines. If this is not possible we can try to apply some iterative search method.

  Another constructive constraint solver quite useful for this purpose is Fudos' bottom-up method [4]. In their method, a cluster corresponds to a well constrained system, and behaves like a rigid body which can only be translated or rotated. Therefore, to maintain a constraint network, clusters detected in the dependency graphs need not be evaluated again.

- Numerical constraint solver:

  In numerical constraint solvers, the constraint network are satisfied by first translating it into a system of equations and then the system is solved by iterative method such as Newton-Raphson method.

  For example, a distance constraint between two points, (X_1, Y_1) and (X_2, Y_2), can be translated into the following equation:

$$(X_1 - X_2)^2 + (Y_1 - Y_2)^2 - D^2 = 0$$

  After translating the dependency graph shown in Figure 5 (c) into a system of equations, we get three equations derived from three distance constraints respectively, and four variables representing the coordinates of point C and point D. As before, one degree of freedom of point C needs to be fixed interactively. Finally, we have a total of four variables and four equations, which can generally be solved by Newton-Raphson iteration.

  One advantage of using numerical methods in this application is that the constraint networks were already satisfied, and each time, we only make a small perturbation to the previous states. Therefore, we will not have the problem of coming up with good initial guesses. However, the disadvantage is that while reevaluating the states of the constraint networks incrementally, we will, from time to time, come across states which make the Jacobian matrices ill-defined.

We experimented with both, constructive, and numerical methods in our implementation of the algorithm. We concluded that a constructive, geometric method is very efficient and robust in those cases where an analytic solution can be found. We plan to extend this method by adding an iterative component which approximates solutions if no analytic solution can be found. The degree of freedom analysis step helps making the constraint subsystem fully constrained, and also in coming up with an evaluation plan.

6 Extensions of the Approach

In this section, we describe several extensions and variations to the basic algorithm.

6.1 Representing Parameters as Objects

In this section we propose to represent parameters of constraints explicitly as an object in the constraint network. Figure 7 shows the representation of a distance constraint with the parameter. Scalar parameters own one degree of freedom. If the parameter represents a constraint it's degree

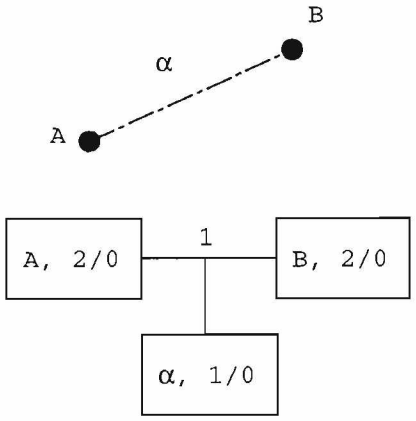


Figure 7: Constraint network with parameters.

of freedom is consumed locally. Without locally constraining the parameter, adding the distance constraint would not change the overall degree of freedom. To change the value of the parameter in this representation, we only need to make the node representing that parameter the root node, and request one degree of freedom from it. This principle can be used to implement an incremental constraint solver that finds a solution, everytime a new constraint is added. Once the constraint is established, the value of the parameter is fixed by a local constraint.

Using parameter objects, we can also represent *congruence relations*, i.e. equality relations between parameters of the constraints. Whenever the parameters of different constraints are set to be equal, they will share the single object that represents the parameter.

6.2 Algebraic Relations

Using the parameter representation described above, the degrees-of-freedom analysis can also be extended to handle *algebraic relations*, between variables or between the parameters of the constraints. Algebraic operations '+' and '×' are introduced as relations with valency one between parameters.

Let's first take a look at an example involving algebraic relations only. The graph in figure 8 (a) for example, shows a linear equation between 5 variables. To change C, a propagation method such as retraction will come up with a plan like the one shown in figure 8 (b). It says that once we get a new value for C, we can use A and C to deduce B, and use C and E to deduce D. Transforming the same problem into our representation, we obtain a constraint network as shown in figure 8 (c). Figure 8 (d) shows a dependency graph that will lead to an evaluation plan equivalent to the plan constructed by the retraction approach.

Next, we will look at an example involving both geometric and algebraic relations. Figure 9 (a) shows a symmetric triangle with an additional algebraic relation defined on the three sides. The algebraic relation is

$$\alpha + \beta = \gamma$$

where γ is a constant. Point A is fixed in space. When point B is dragged, a dependency graph is constructed as shown in Figure 9. An evaluation plan for a constructive constraint solver can be set up to maintain the triangle:

1. Point B is assigned the coordinates of the locator device.

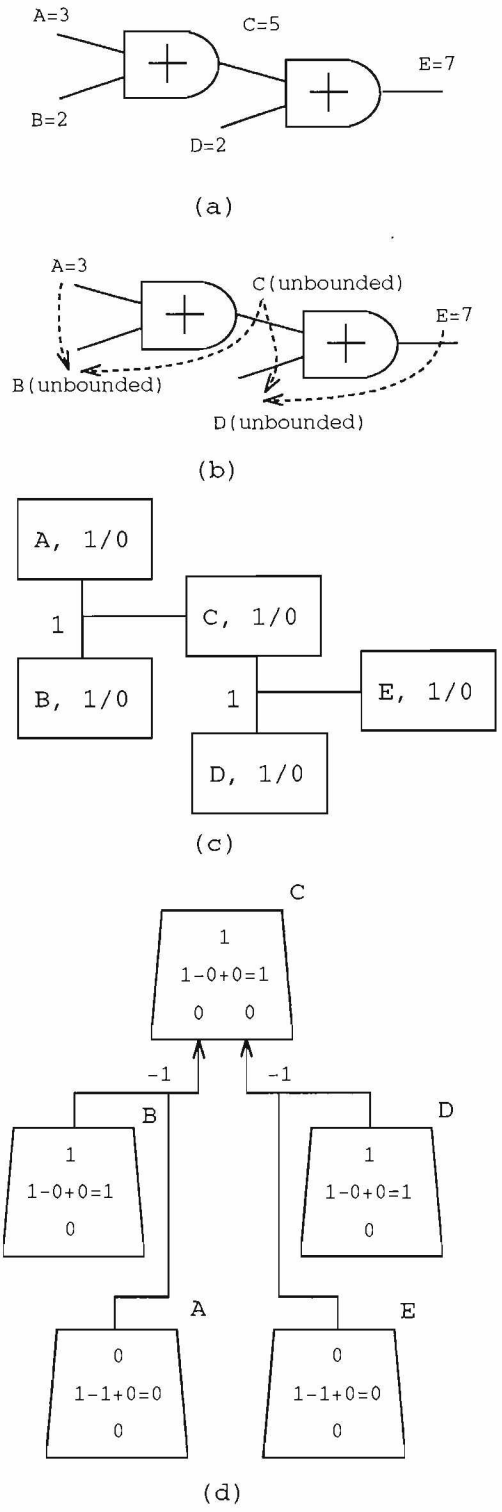
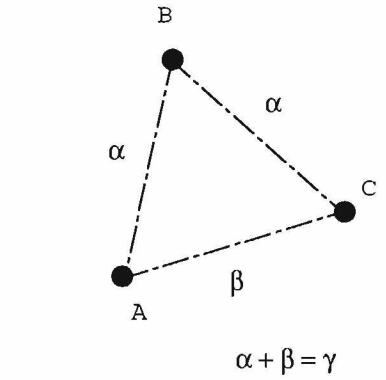
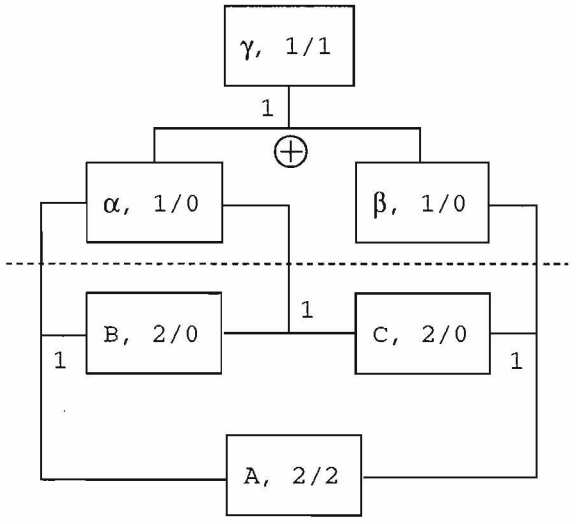


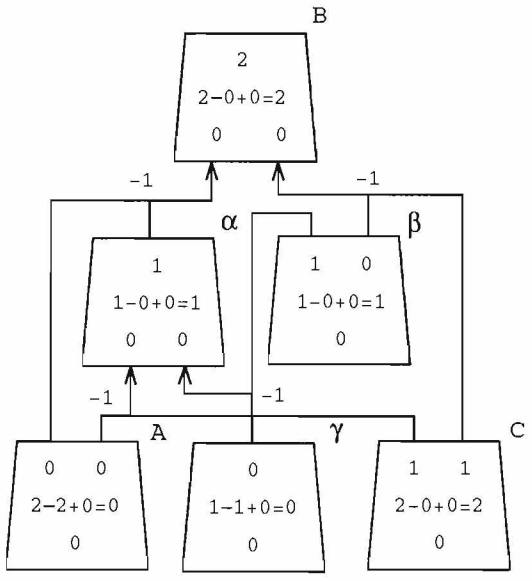
Figure 8: An example of algebraic constraints.



(a) The model



(b) The constraint network



(c) The dependency graph

Figure 9: A symmetric triangle with algebraic constraints.

2. Parameter α is equal to the distance between point A and new point B.

3. Parameter β is equal to γ - α.

4. Point C is determined by intersecting the two circles centered at new point B and point A respectively with the radii equal to the new parameters.

7 Constraints in 3-D

In this section, we describe an interactive 3-D constraint system that supports polyhedron definition and manipulation. A polyhedron is made up of half planes, edges, and vertices. A repertoire of constraints can be defined on these geometric objects. We will show how the same degrees-of-freedom analysis algorithm can be used to deal with interactive manipulation of 3-D models.

7.1 Boundary Representations as Constraint Networks

Boundary representations (B-reps) of polyhedra usually consist of faces, edges, and vertices. However, polyhedra could be defined by only one type of primitives, namely half spaces. Edges and vertices are objects derived from intersecting half spaces. A half space is defined by an oriented plane. A point $\vec{p}$ on the plane can be written as:

$$\vec{p} \cdot \vec{n} = \delta \tag{14}$$

A half space in 3-D has two rotational and one translational degrees of freedom. The end points of all the unit normal vectors will fall on the surface of the unit sphere centered at the origin. We will refer to the point as the *orientation point* of the corresponding half plane.

An edge in a B-rep is derived by intersecting two planes. We can represent the incidence relation between edges and planes by constraints, as shown in figure 10 (a).

A vertex can be derived from three intersecting planes, or by intersecting one edge and one plane, as shown in figure 10 (b) and (c).

Note that the total number of degrees of freedom stored in the constraint network is unchanged by the introduction of the derived objects, since they are totally dependent on the half spaces.

Representing a derived object is sometimes useful, for instance, when a constraint is defined on it. There are several types of constraints that are defined on derived objects. For example, a distance constraints can be defined between a plane and a vertex, or between two vertices as shown in figure 11.

Figure 12 shows an example polyhedron (a block, with a notch cut out). Suppose that vertices v4, and v6 are fixed in space. If we grab vertex v1 and request one degree of freedom, the degree-of-freedom analysis algorithm will set up a dependency graph as shown in figure 13. The dependency graph indicates that in order to move vertex v1 with one degree of freedom, we have to change plane p3 with one degree of freedom, while keeping p1 and p2 fixed. Since v4 and v6 are fixed, plane p3 will rotate around the axis through v4 and v6. In addition, vertex v2 which is on p3 will move along the edge derived from intersecting planes p2 and p5. Similarly, vertex v3 will move along the edge derived from intersecting planes p2 and p6. Figure 14 shows the snapshots taken from the interactive system, running this example problem. The execution time to build the dependency

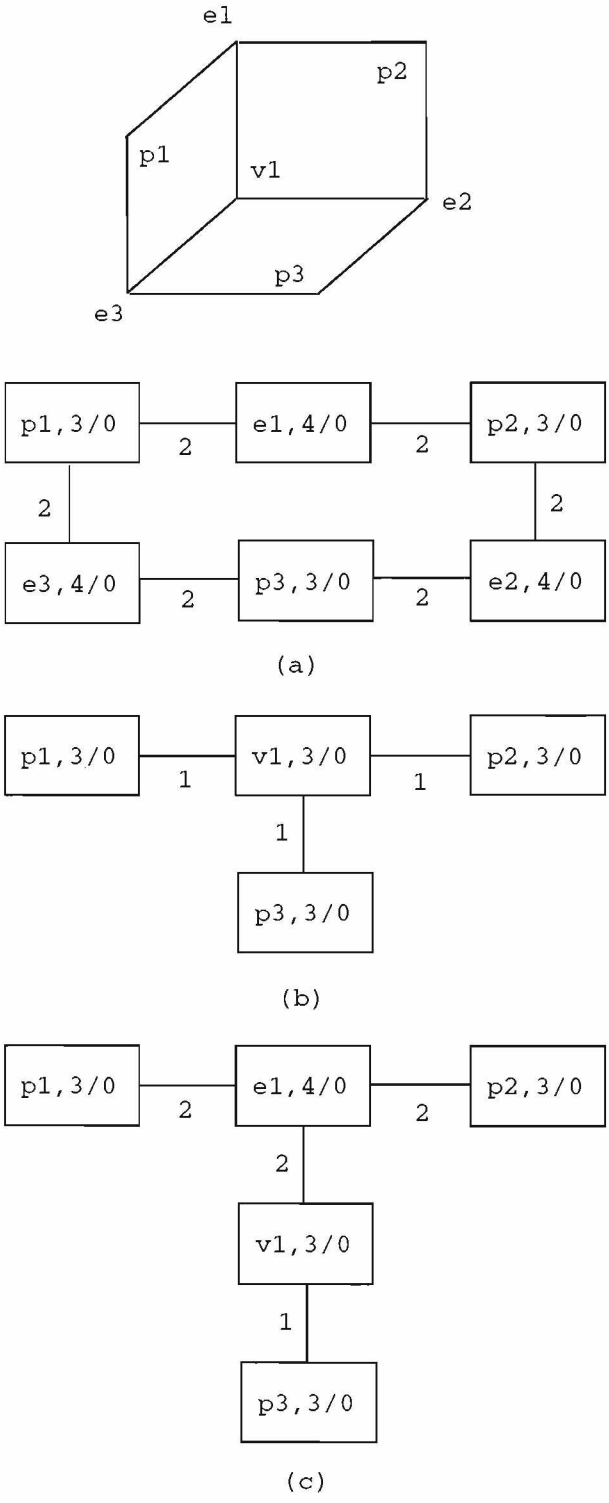


Figure 10: Incidence relation between planes, edges, and vertices.

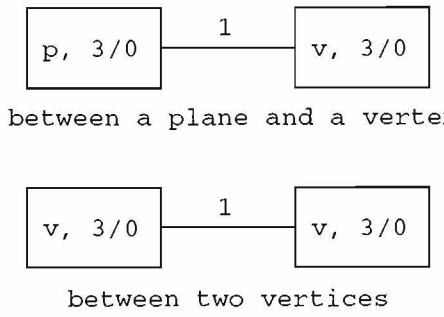


Figure 11: Various distance constraint types.

graph was about 1/20 second on a Sparc-10. The evaluation of the symbolic solution is done in real time. Note that all the incidence constraints between the planes and vertices were derived automatically from the boundary representation, by the system; only the two position constraints were added interactively.

An interesting observation is that there is a correspondence between the dependency graph found by the algorithm, and the resolvable sequence defined in Sugihara's paper [29]. One of the resolvable sequences that can be defined for the model above is:

($\cdots$ p1 $\cdots$ p2 $\cdots$ p5 $\cdots$ p6 $\cdots$ v4 $\cdots$ v6 $\cdots$ v1 p3 v2 v3)

In this partial sequence, we observe that all the objects that are fixed in the dependency graph appear before the vertex v1. Vertex v1 is placed after plane p1 and p2; therefore it has one degree of freedom along the intersecting edge. Once we placed v1, we can place p3, v2, and v3 as shown in the resolvable sequence.

In the next example, we added distance constraints between vertices v1 and v2, v2 and v5, v5 and v3, v3 and v4. The construction sequence found by the solver involves intersections between spheres and planes. The behavior, when dragging vertex v1 is quite different to before. Two snapshots of the interactive session are shown in figure 15.

With the added constraints, the system took about 10 seconds to derive the symbolic solution in form of a dependency graph. Again the evaluation can be done at interactive speeds. There is indication, that the execution time for the symbolic part of the constraint solution grows exponentially with the number of constraints involved (due to the non-deterministic nature of the search). This behavior is typical for all symbolic algebraic constraint solvers (for instance Grobner bases or the resultant method), and there is evidence that a polynomial complexity cannot be achieved, except in very restricted problem domains. This is usually not a problem for two dimensional problems, or for mostly underconstrained situations. However, in three dimensions, and for well constrained problems this might become prohibitive. Once the symbolic solution is found, it can be evaluated in linear time, however. Fortunately, due to the geometric nature of the constraint solver, it can be integrated well with other geometric construction operations. The idea is to use constructors and dependencies most of the time, and to add only a few dimensions as constraints, rather than expressing everything in a constraint context. Also, there are many possibilities for preprocessing and for speeding up the constraint solving which haven't been explored, yet.

The degree of freedom analysis algorithm works well in the above case. However, if we fix vertices v3 and v4 instead, the algorithm will possibly construct a dependency

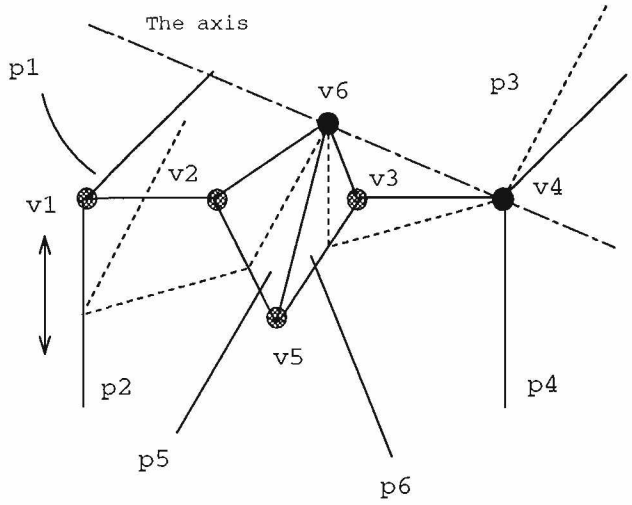


Figure 12: A polyhedron with notch.

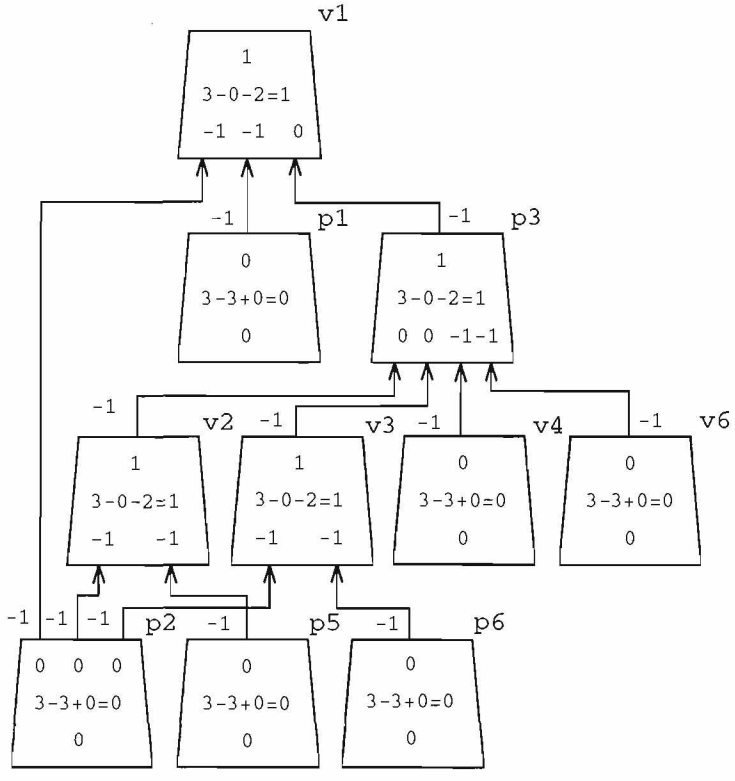


Figure 13: A dependency graph for moving v1 in one degree of freedom.

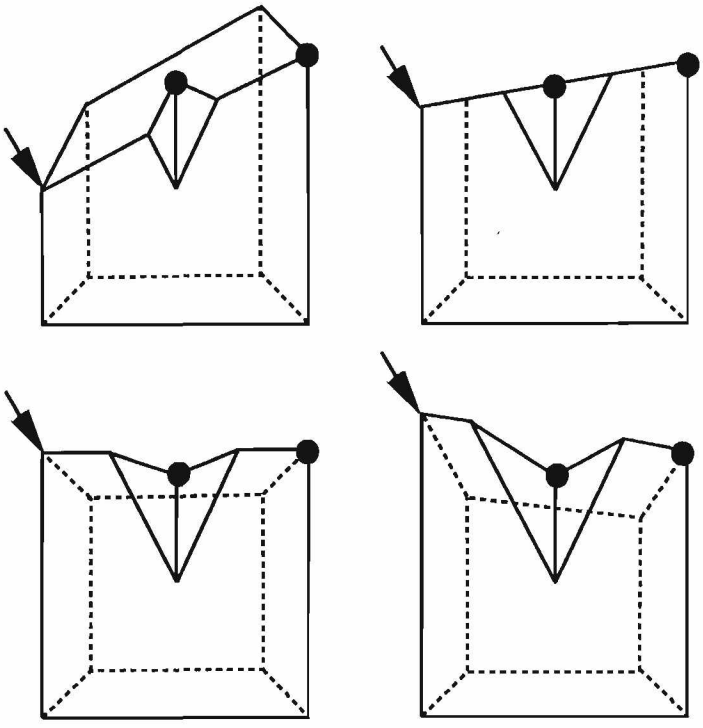


Figure 14: Snapshots taken while interactively dragging vertex v1.

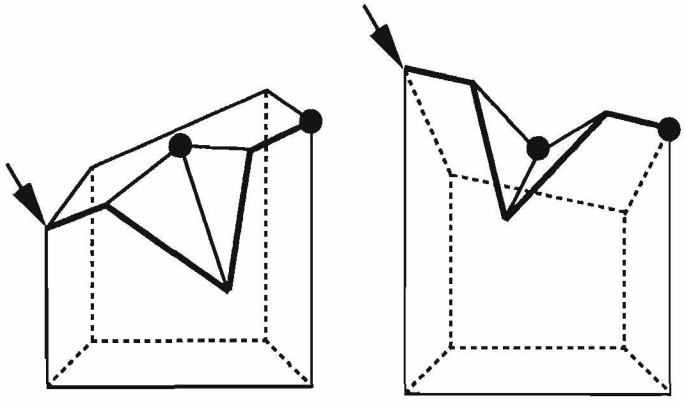


Figure 15: More snapshots, after distance constraints were added.

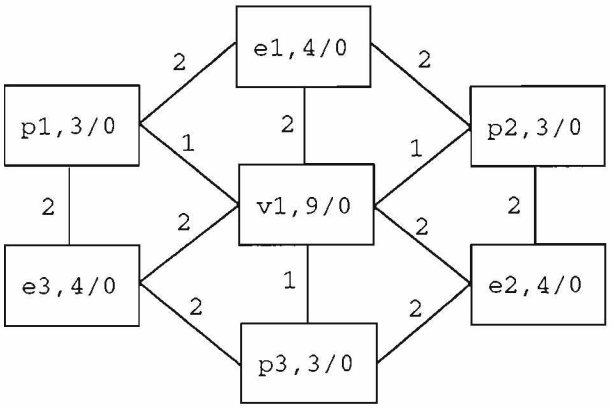


Figure 16: A representation with redundant degrees of freedom and valencies.

graph similar to the one above (by simply exchanging the labels v3 and v6 in figure 13), but it will fail to evaluate it. The reason is that the algorithm assumes general positions for all objects, and hence it is unable to recognize the special case where vertices v1, v3, and v4 are collinear. The proposed solution, namely to rotate p3 about an axis through v3 and v4 does not yield the desired degree of freedom for v1. One remedy is to represent the edges explicitly. This way, the analysis algorithm will realize that v1 is incident on an edge that is completely constrained by v3 and v4, and the only degree of freedom v1 has, is along this edge. Through backtracking it will produce a dependency graph that will move half plane p1 in one degree of freedom and fix half plane p3. However, this approach raises another difficulty. Representing all the incidences between edges and vertices is redundant, and will lead to an over-constrained system. On the other hand, since the redundant constraints are always consistently over-constrained we may fix the problem by artificially increasing the degrees of freedom of each vertex to compensate for the redundancy. Figure 16 shows a subset of the constraint network around vertex v1 (points are 9-dimensional, edges are 4-dimensional).

8 Conclusion

The graph based algorithm presented in this paper is a very general tool for reasoning about constraint systems. The algorithm is independent of the dimension of the space. We have presented a number of 2-D examples as well as an application to 3-D space. The method works for both, geometric constraint systems and algebraic ones.

A variety of symbolic or numerical techniques can be used in the evaluation part. We chose to implement a geometric, constructive method, which seeks to find solutions by intersections of geometric objects, such as lines, circles, planes, spheres, etc. A construction plan is derived directly from the dependency graph. An iteration mechanism could also find solutions that cannot be directly computed symbolically.

The approach may prove to be a powerful tool in interactive geometric modeling, especially in 3D, with newly emerging virtuality devices, such as gloves, and bats. The success, however, will hinge upon powerful evaluation methods, and powerful user interfaces techniques. The framework given here lays the theoretical foundations. The prototype implementation shows that the approach is working quite well in interactive situations.

With the approach taken here we provide the capability to simulate the degrees of freedom of under-constrained networks of constraints which enable user to design in a less restricted way. Users are not forced to specify shapes completely by constraints but can freely mix constraint definitions with geometric constructions in a more intuitive way.

Acknowledgments

Thanks to Greg Alt for his work on the graphical user interface.

References

[1] AHO, A., HOPCROFT, J., AND ULLMAN, J. *Data Structures and Algorithms.* Addison-Wesley, 1983.

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

[3] BORNING, A. H. The programming language aspects of thinglab, a constraint-oriented simulation laboratory. *ACM Transactions on Programming Languages and Systems 3*, 4 (October 1981), 353–387.

[4] BOUMA, W., FUDOS, I., AND HOFFMANN, C. A geometric constraint solver. Technical Report CSD-TR-93-054, Department of Computer Science, Purdue University, 1993.

[5] BRÜDERLIN, B. Constructing three-dimensional geometric object defined by constraints. In *Proceedings of the 1986 Workshop on Interactive 3D Graphics, ACM SIGGRAPH* (Chapel Hill, North Carolina, 1986).

[6] BRÜDERLIN, B. *Rule-Based Geometric Modelling.* PhD thesis, ETH Zürich, Switzerland, 1987.

[7] BRÜDERLIN, B. Using geometric rewrite rules for solving geometric problems symbolically. *Theoretical Computer Science 2*, 116 (August 1993), 291–303.

[8] FREEMAN-BENSON, B. N., AND MALONEY, J. The deltablue algorithm: An incremental constraint hierarchy solver. Technical Report 88-11-09, Computer Science Department, University of Washington, November 1988.

[9] FREEMAN-BENSON, B. N., MALONEY, J., AND BORNING, A. An incremental constraint solver. *Communications of the ACM 33*, 1 (January 1990), 54–63.

[10] FUDOS, I., AND HOFFMANN, C. Correctness proof of a geometric constraint solver. Technical Report CSD-TR-93-076, Department of Computer Science, Purdue University, December 1993.

[11] GOSLING, J. Algebraic constraints. Technical Report CMU-CS-83-132, Carnegie-Mellon University, 1983.

[12] HILLYARD, R., AND BRAID, I. Analysis of dimensions and tolerances in computer-aided mechanism design. *Computer Aided Design 10*, 3 (May 1978), 161–166.

[13] HILLYARD, R., AND BRAID, I. Characterizing non-ideal shapes in terms of dimensions and tolerances. *Computer Graphics 12*, 3 (1978), 234–238.

[14] HOFFMANN, C., AND VERMEER, P. Geometric constraint solving in r^2 and r^3. In *Computing in Euclidean Geometry, 2nd Edition.* World Scientific Publishing Co Pte Ltd, 1994.

[15] HSU, C., AND BRÜDERLIN, B. Constraint objects - integrating constrinat definition and graphical interaction. In *Proceedings of the 1993 ACM/SIGGRAPH Symposium on Solid Modeling Foundations and CAD/CAM Applications* (Montreal, Canada, May 19-21 1993).

[16] HSU, C., AND BRÜDERLIN, B. Constraint objects - integrating constrinat definition and graphical interaction. Technical Report UUCS-93-019, Computer Science Department, University of Utah, 1993.

[17] HSU, C., AND BRÜDERLIN, B. Moving into higher dimensions of geometric constraint solving. Technical Report UUCS-94-027, Computer Science Department, University of Utah, 1994.

[18] JR., G. S., AND SUSSMAN, G. Constraints - a language for expressing almost-hierarchical descriptions. *Artificial Intelligence 14*, 1 (January 1980), 1–39.

[19] JUSTER, N. Modelling and representation of dimensions and tolerances: a survey. *Computer Aided Design 24*, 1 (1992), 3–17.

[20] KRAMER, G. A. Using degrees of freedom analysis to solve geometric constraint systems. In *Proceedings of the 1991 ACM/SIGGRAPH Symposium on Solid Modeling Foundations and CAD/CAM Applications* (New York, 1991), ACM Press.

[21] LELER, W. *Constraint Programming Languages: Their Specification and Generation.* Addison-Wesley Publishing Company, Inc, 1988.

[22] LIGHT, R., AND GOSSARD, D. Modification of geometric models through variational geometry. *Computer Aided Design 14*, 4 (July 1982), 209–214.

[23] LIN, V., GOSSARD, D., AND LIGHT, R. Variational geometry in computer-aided design. *Computer Graphics 15*, 3 (August 1981), 171–177.

[24] OWEN, J. Algebraic solution for geometry from dimensional constraints. In *Proceedings of the 1991 ACM/SIGGRAPH Symposium on Solid Modeling Foundations and CAD/CAM Applications* (May 1991).

[25] ROLLER, D. An approach to computer-aided parametric design. *Computer Aided Design 23*, 5 (June 1991), 385–391.

[26] ROLLER, D., SCHONEK, F., AND VERROUST, A. Dimension-driven geometry in cad: A survey. In *Theory and Practice of Geometric Modeling.* Springer Verlag, 1989, pp. 509–523.

[27] ROY, U., LIU, C., AND WOO, T. Review of dimensioning and tolerancing: representation and processing. *Computer Aided Design 23*, 7 (1991), 466–483.

[28] SERRANO, D., AND GOSSARD, D. Combining mathematical models and geometric models in cae systems. In *Proc. ASME Computers in Eng. Conf.* (Chicago, July 1986), pp. 277–284.

[29] SUGIHARA, K. Resolvable representation of polyhedra. In *Proceedings of the 1993 ACM/SIGGRAPH Symposium on Solid Modeling Foundations and CAD/CAM Applications* (Montreal, Canada, May 19-21 1993).

[30] SUTHERLAND, I. *Sketchpad, a man-machine graphical communication system.* PhD thesis, MIT, January 1963.

[31] VERROUST, A., SCHONECK, F., AND ROLLER, D. Rule-oriented method for parameterized computer-aided design. *Computer Aided Design 24*, 10 (October 1992), 531–540.