Moving into Higher Dimensions of Geometric Constraint Solving

Ching-yao Hsu
Beat Bruderlin

UUCS-94-027

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

November 1, 1994

Abstract

In this paper, we present an approach to geometric constraint solving, based on degree of freedom analysis. Any geometric primitive (point, line, circle, plane, etc.) possesses an intrinsic degree of freedom in its embedding space which is usually two or three dimensional. Constraints reduce the degrees of freedom of an object (or a set of objects). We use graph algorithms to determine upper and lower bounds for the degrees of freedom of a set of constrained objects, symbolically. This analysis is then used to establish dependency graphs and evaluation schemes for symbolic or numeric solutions to constraint problems. The approach has been generalized for 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. Also, higher than three dimensional solutions can be interpreted as approaches to over- and under- constrained problems. In this paper, we will present the theoretical background of the approach, and demonstrate how it can be applied within an interactive design environment.

Moving into Higher Dimensions of Geometric Constraint Solving

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

Abstract

In this paper, we present an approach to geometric constraint solving, based on degree of freedom analysis. Any geometric primitive (point, line, circle, plane, etc.) possesses an intrinsic degree of freedom in its embedding space which is usually two or three dimensional. Constraints reduce the degrees of freedom of an object (or a set of objects). We use graph algorithms to determine upper and lower bounds for the degrees of freedom of a set of constrained objects, symbolically. This analysis is then used to establish dependency graphs and evaluation schemes for symbolic or numeric solutions to constraint problems. The approach has been generalized for 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. Also, higher than three dimensional solutions can be interpreted as approaches to over- and under- constrained problems. In this paper, we will present the theoretical background of the approach, and demonstrate how it can be applied within an interactive design environment.

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 sometimes 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 in [25, 19, 26].

Geometric constraints have proven 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

1.1.1 Propagation 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. The weakness of this approach is that it can not handle cyclic constraint situations, and hence they are usually coupled with numerical methods as described in the next section.

Ivan Sutherland's Sketchpad [29] was the pioneer system that used interactive computer graphic and constraint system. The system employed a satisfaction mechanism that used propagation of degrees of freedom and propagation of known values. For a review of these methods, please refer to [20]. ThingLab [5, 9, 10] adopted many of the ideas from Sketchpad used a constraint hierarchy together with an incremental constraint solver, DeltaBlue. DeltaBlue is fast because it is a local propagation algorithm. CONSTRAINTS [18] was based on a technique called retraction. It is essentially a localized version of propagation of known values. Information of flow during propagation is stored in the variables as premises

and dependents. Subsequent constraint satisfaction can take advantage of these previous flow patterns to achieve good performance. However, it is in general less powerful than propagation of known values. To alleviate the cyclic constraint situations, Magritte [12] utilizes a mechanism which transforms a set of constraints into a single constraint algebraically. Breadth-first planning is then used to search for all solutions. The transformation technique reduces the number of situations where a numerical method is needed, but does not completely eliminate them.

1.1.2 Numerical Methods

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. 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. Relaxation works by perturbing the values assigned to variables in a way that the total error is minimized. The problem with relaxation is that its convergence is quite slow.

The Newton-Raphson method is another popular numerical method. It is much faster than relaxation but the solution also depends heavily on the initial guesses. It is the basic technique used in the variational geometry approach [14, 15, 22, 21, 27]. In [4], a projection method was introduced to find solutions that minimize the Euclidean distance between the new and old solutions. It can work with under-constrained systems and the solution presented may be more predictable. In [31], constraints are defined as energy functions, and the solution is obtained by minimizing the energy. [13] proposes a new idea of soft constraints solving strategy, that could lead the way in integrating constraints and tolerances. The solving of their system is based on a technique developed in control theory, the Kalman filter [3].

1.1.3 Constructive 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 evaluation is determined, and can be roughly divided into two categories, the rule-based approach and the graph-based approach.

The rule-based approach is characterized by using rewrite rules to derive the sequence of construction steps. In earlier research, Brüderlin developed a new mechanism for symbolic geometric constraint solving [7, 8]. In the mechanism, constraints are represented symbolically as predicates over points. A rewrite rule mechanism seeks to match the left hand side of a rule with a subset of the constraints. If a rule applies some of the predicates are replaced by new, simpler ones, and a construction operation is applied to satisfy these constraints simultaneously. Aldefeld [2], Sunde, and Roller [24, 30] have also proposed rule-based approach for geometric constraint solving.

Another approach by Owen [23] is a graph-based algorithm which finds a solution for all completely and consistently constrained definitions, if there exists a solution that can be constructed by ruler and compasses. The algorithm is composed of two phases. In the top-down phase the graph is analyzed and a sequence of construction step is derived. In the bottom-up phase, the construction steps are carried out and the model is constructed. Fudos [6] developed an approach similar to Owen's. The main difference between the above two approaches is that Owen's solver is top-down as described above, whereas Fudos' solver works bottom-up, placing first the geometric elements into clusters, and then coalescing clusters. A comprehensive correctness proof is given in his paper [11] as well.

1.2 An Overview

In most constraint-based geometric modeling system, the problem concerned is, given a complete and consistent set of constraint specification, how to get a model evaluated automatically. However, coming up with a consistent specification is not an easy task, especially in the early design stage when user does not have a clear picture of what the model is really like.

Therefore, it is our objective to develop an approach to provide the capability to simulate 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), and is capable of extracting a part with specified degrees of freedom from a constraint network. The constraint network can then be manipulated through the degrees of freedom acquired.

The rest of the paper is organized in the following manner:

- 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 to maintain the constraint network.
- Section 6 presents several variations and extensions to the basic algorithm which are useful for building an effective user interface.
- Finally, 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 $O = \{ o_1, o_2, \ldots, o_n \}$ 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 $DOF = \{ dof_1, dof_2, \ldots, dof_l \}$ of degrees of freedom owned by objects in O completely determine the *set of states* of the geometric model.

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.

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 is a positive number which is less than or equal to the sum of the degrees of freedom owned by the n objects.

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

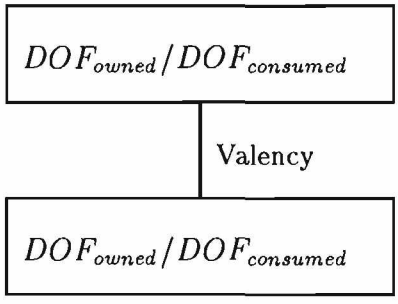
where $DOF(o_{i_j})$ refers to the degrees of freedom of the object o_{i_j}, and $valency(c_i)$ denotes the valency of the constraint c_i. If the valency is equal to the total number of the degrees of freedom owned by the n objects, we call the constraint a *full-valency constraint.* A full-valency constraint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

We also define separately a special class of constraints, *local constraints*, which are unary constraints used to *consume* 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.*

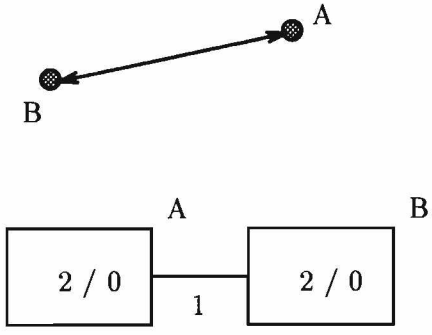
2.2 Constraint Network

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

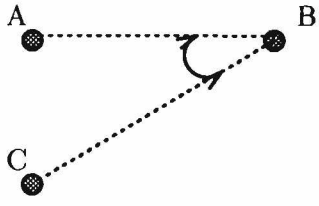
A *constraint network* is an undirected graph which consists of a finite set of nodes and a

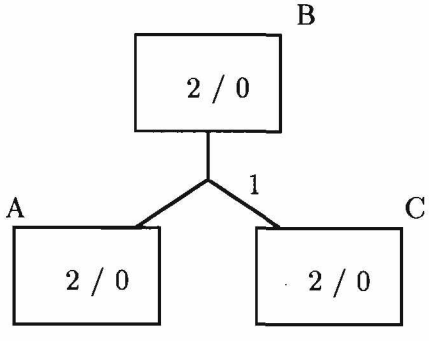


(a) The constraint network representation

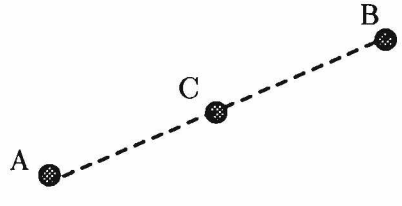


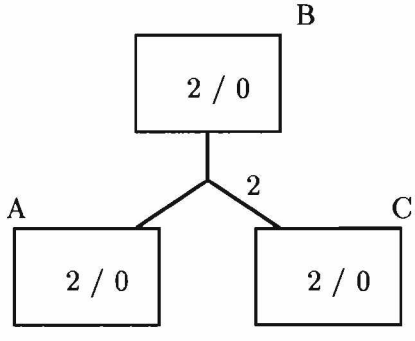
(b) An example of distance constraint





(c) An example of Angle constraint





(d) An example of midpoint constraint

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

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} is the degrees of freedom owned by the unconstrained object, whereas the second number, $DOF_{consumed}$ is the degrees of freedom consumed by the local constraints, if there are any. An *arc* represents an n-ary relation and is therefore a general 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.

Suppose that 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 that of the valencies. On the other hand, to fully 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 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.

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

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

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

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

[1]Please refer to [16, 17] for a similar algorithm developed from a 2-D geometric point of view.

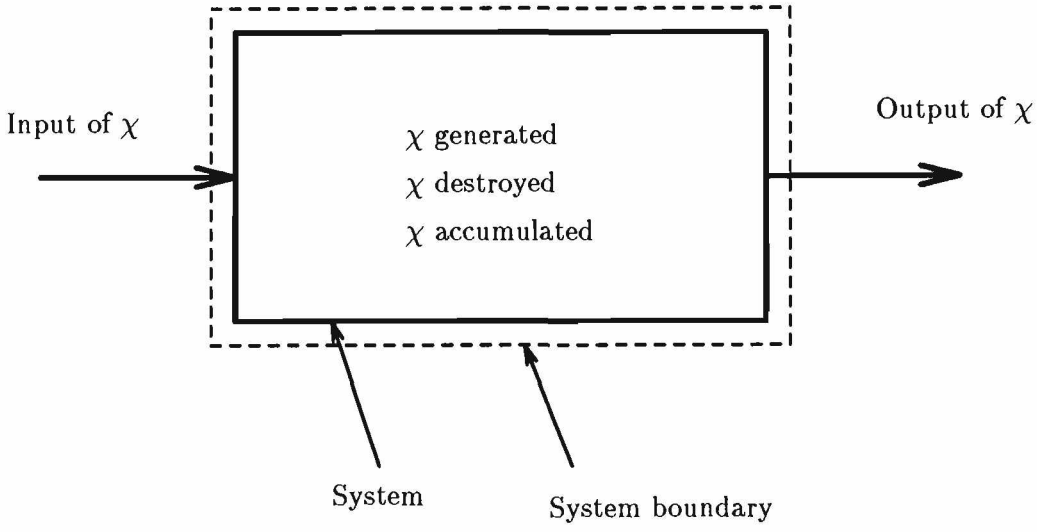


Figure 2: A system with input and output.

Suppose that we are interested in the balance of some quantity, say χ with respect to a system as shown in Figure 2. χ can either enter the system or leave the system. It can also be generated or destroyed within the system. If the conservation law holds for χ, then

$$\left[\begin{array}{c}\text{rate of } \chi \\ \textit{output}\end{array}\right] = \left[\begin{array}{c}\text{rate of } \chi \\ \textit{input}\end{array}\right] + \left[\begin{array}{c}\text{rate of } \chi \\ \textit{generated}\end{array}\right] - \left[\begin{array}{c}\text{rate of } \chi \\ \textit{destroyed}\end{array}\right] - \left[\begin{array}{c}\text{rate of } \chi \\ \textit{accmulated}\end{array}\right] \quad (3)$$

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 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, equation3 can be shortened to:

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

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 obey the balance equation for degrees of freedom. 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, we have for a node (see also Figure 3),

$$\begin{aligned} DOF_{out} &= DOF_{in} + DOF_{owned} - DOF_{consumed} \\ DOF_{in} &= \sum_i DOF_{in}(i) \\ DOF_{out} &= \sum_j DOF_{out}(j) \end{aligned} \tag{5}$$

For an arc (see also Figure 4),

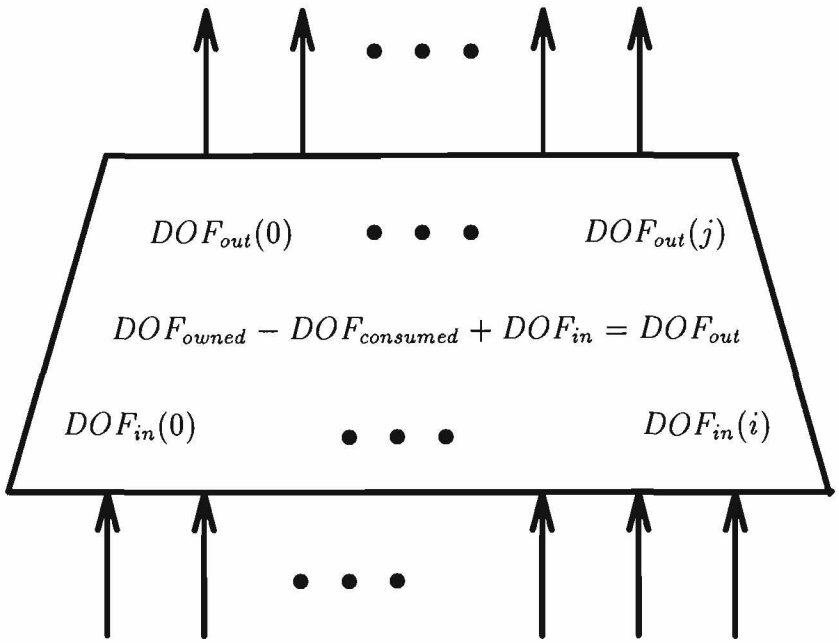
$$\begin{aligned} DOF_{out} &= DOF_{in} - valency \\ DOF_{in} &= \sum_k DOF_{in}(k) \end{aligned} \tag{6}$$

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.

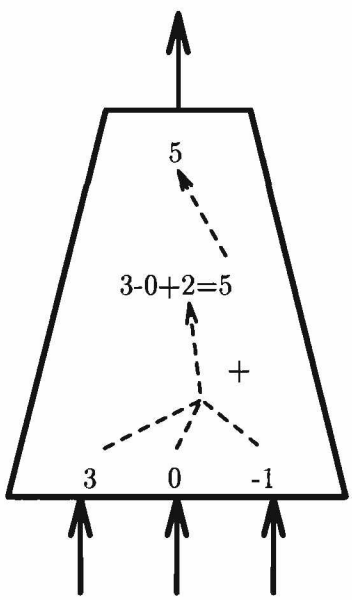
3.2.1 An Example

Figure 5 (a) shows a simple geometric model consisting of four points and three distance constraints in 2-D space. Figure 5 (b) shows the corresponding constraint network and Figure 5 (c) shows one instance of the dependency graph derived from it.

As shown in the (c) part of the picture, there are no degrees of freedom entering the nodes at the lowest level of each branch and all the degrees of freedom owned by the nodes are consumed. As a result, there is zero degrees of freedom at the output of these nodes.

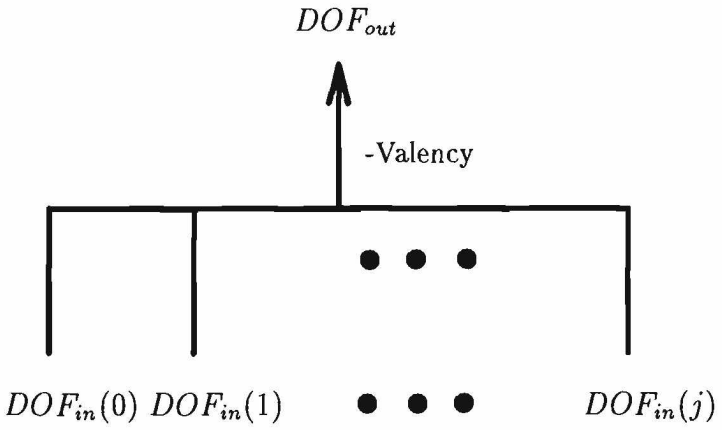


(a)

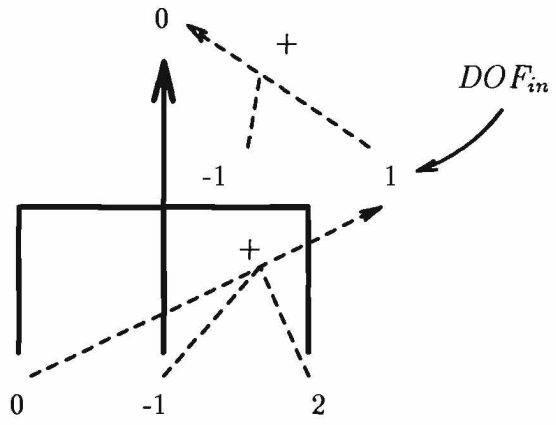


(b)

Figure 3: The node of a dependency graph.



(a)



(b)

Figure 4: The arc of a dependency graph.

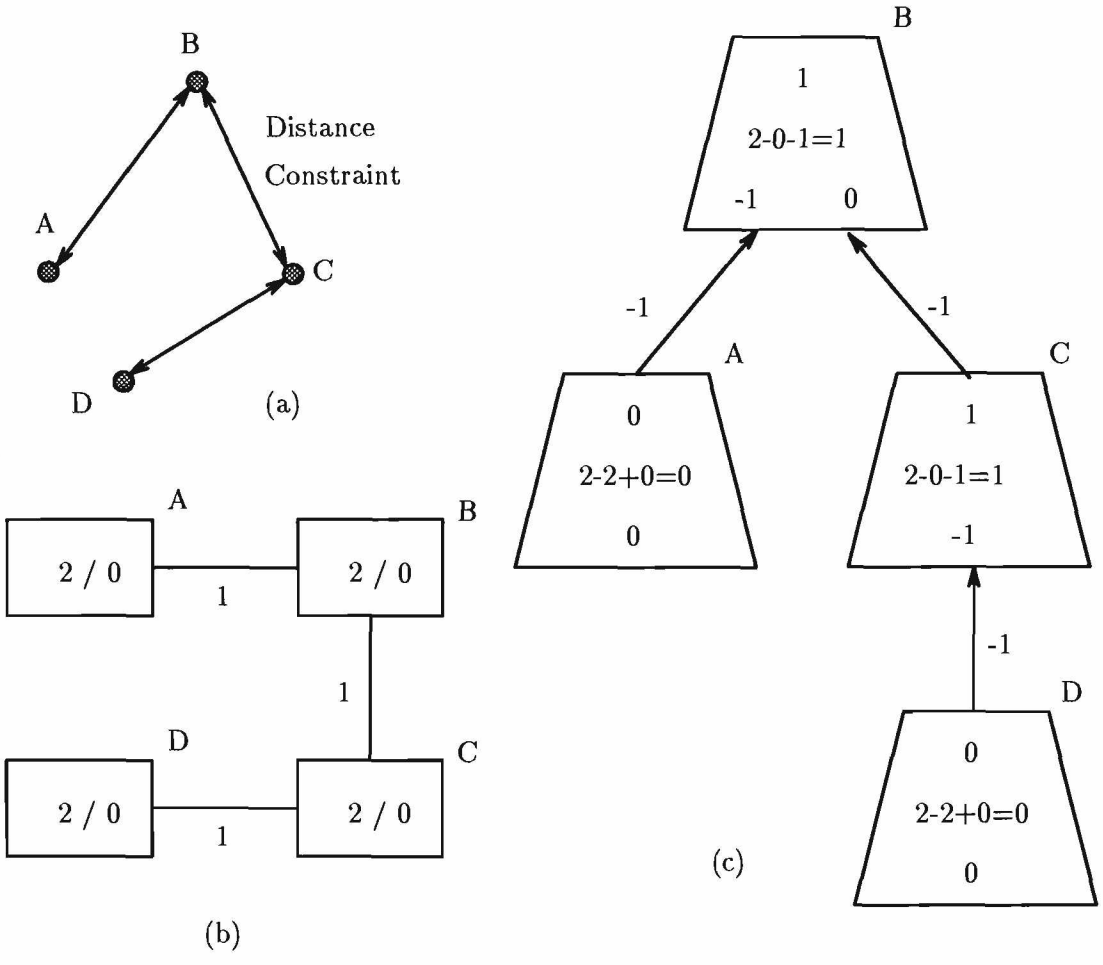


Figure 5: An instance of the dependency graph.

In the next level up, one degree of freedom is offset by each constraint, which results in a negative degrees of freedom entering the parent nodes. We finally obtain one degree of freedom out of the topmost node, the *root node.*

This illustrates the balance of the flow of degrees of freedom for a dependency graph in a bottom-up fashion.

3.3 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 want to get from the root 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. Figure 6 shows an invalid dependency graph that was built as a result of a depth-first approach.

3.3.1 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:

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

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.

Suppose that m constraints, $c_1, c_2, \ldots, c_m$ are attached to this object, we then expand

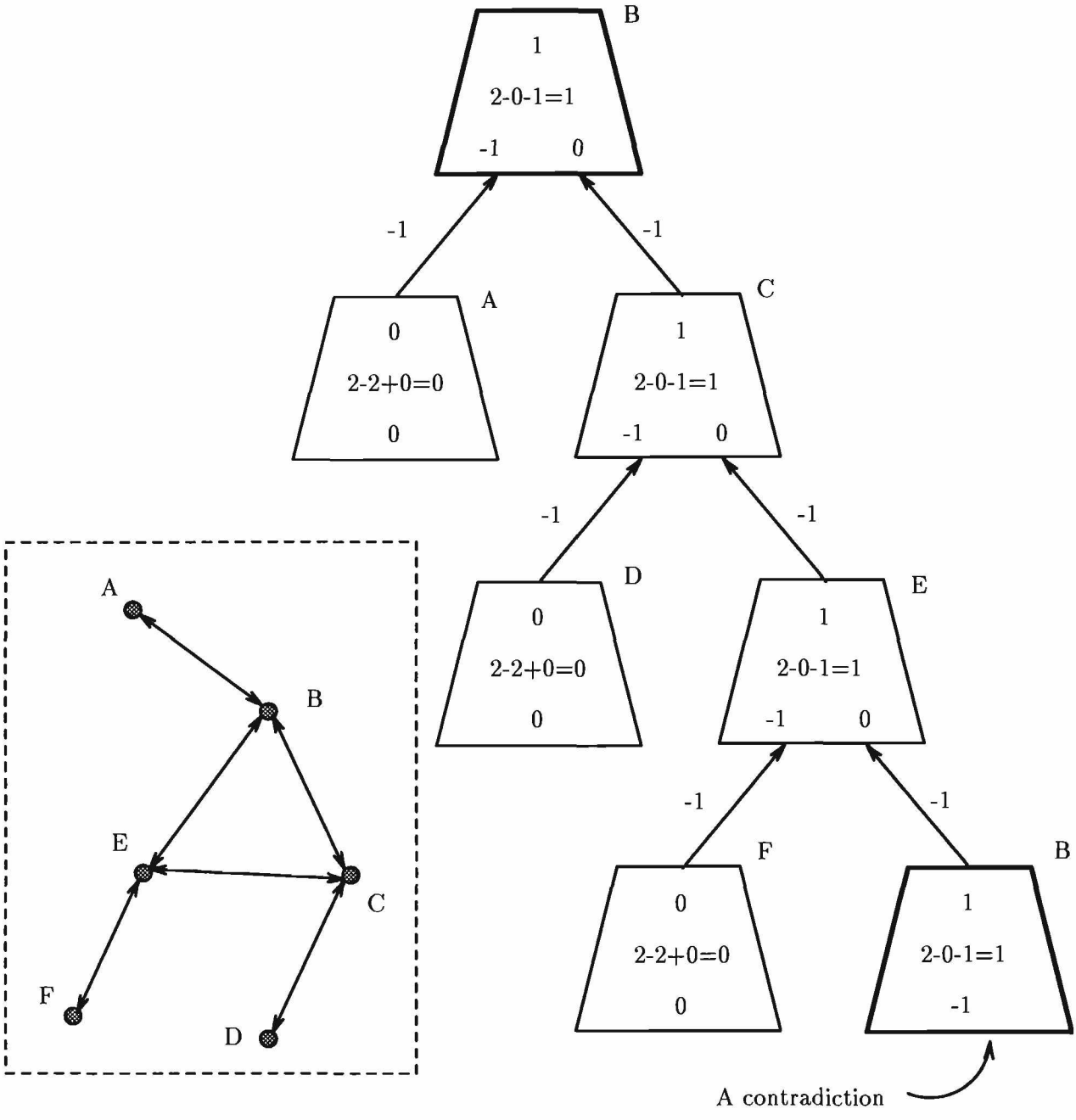


Figure 6: An invalid dependency graph by depth-first approach.

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 been used elsewhere.

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 it is the second condition, we also set $DOF_{consumed}$ equal to DOF_{owned} (i.e. to locally fix the state of the object).

3.3.2 An Example

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

[2]This notion will be made clear later.

- requesting -1 from point A and 0 from point C,
- requesting 0 from point A and -1 from point C,
- requesting -2 from point A and 1 from point C, etc.

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

4 Properties of the Algorithm

In this section, we will discuss some details we have left out in the previous section, including the validity of a request for degrees of freedom, and how to divide the degrees of freedom. A lot of other important properties of the algorithm are discussed here as well.

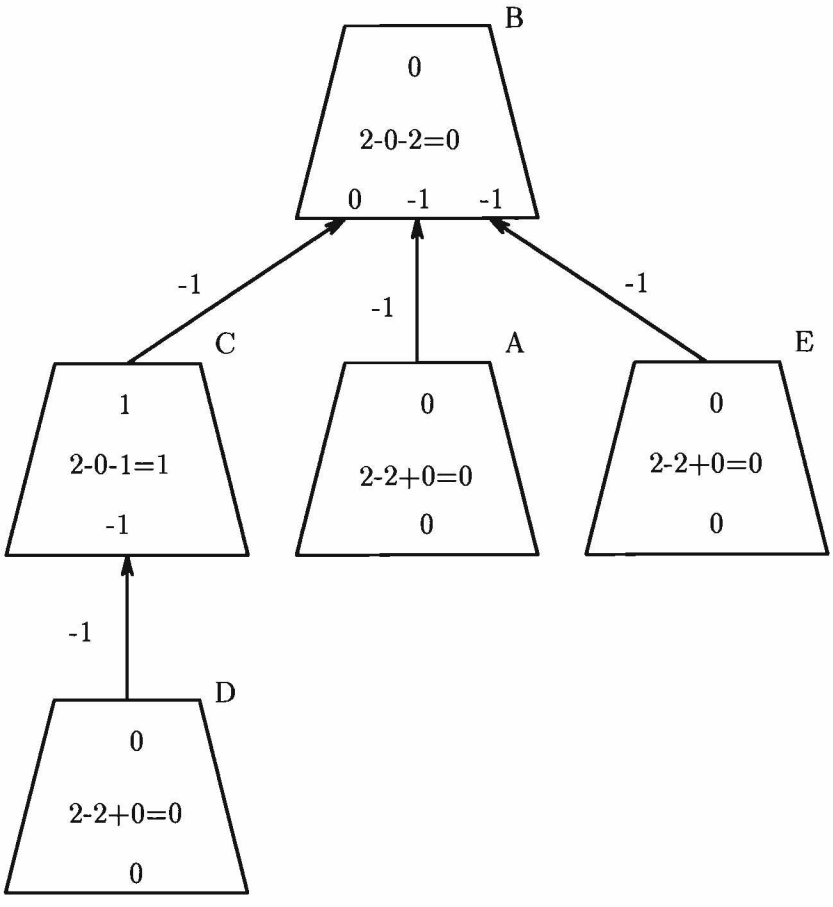
4.1 The Degrees Of Freedom Requested

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.

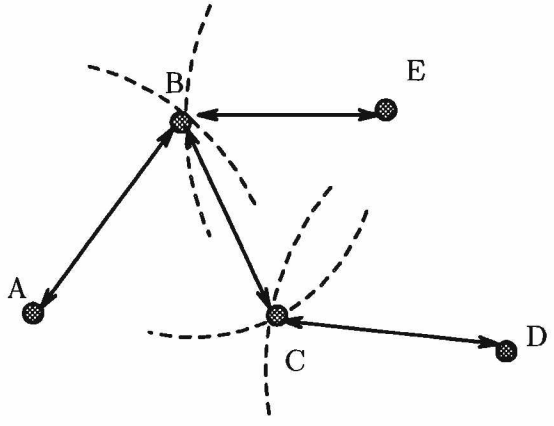
We define the *connected neighborhood of an object o* to be the maximal connected induced subgraph which contains o. Suppose G is the connected neighborhood of the object o. The *maximum attainable degrees of freedom* of the object o is then

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

On the other hand, the lower bound on the degrees of freedom requested is usually 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. 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 7.



(a) The dependency graph



(b) The geometric interpretation

Figure 7: Requesting zero degrees of freedom.

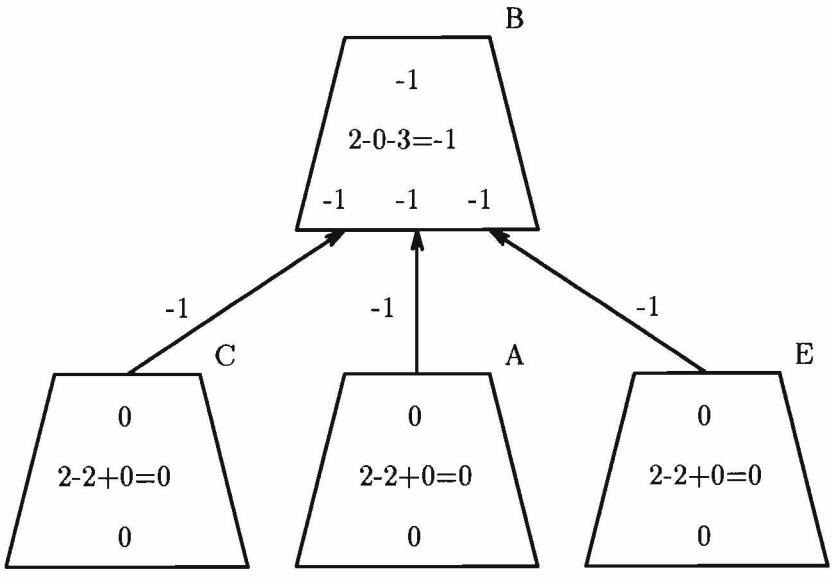


Figure 8: Requesting a negative degree of freedom.

Finally, it is, in fact, possible to construct a dependency graph with a negative degree of freedom as shown in Figure 8. However, the dependency graph represents an over-constrained system and is therefore not useful in common applications.

4.2 On Search Strategies

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 section, we will introduce a division method based on exhaustive search. More sophisticated search methods are possible, but require lengthy treatment. Hence we will not discuss them here.

4.2.1 Exhaustive Search

Here 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 at most zero 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 should not exceed that stored in the object.

$$DOF_{out} \leq DOF_{owned} - DOF_{consumed} \tag{12}$$

  Notice that if $DOF_{out}(j)$ is less than zero, store $DOF_{out}(j)$ away and set $DOF_{out}(j)$ to zero.

- No positive degrees of freedom are imported.

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

These rules also guarantee that the dependency graphs generated are valid. See Figure 9 for an example.

4.3 The Degrees of Freedom Obtained

Assume that the $DOF_{request}$ is valid at all times. The degrees of freedom obtained in a dependency graph depend on the search strategy adopted.

For a dependency graph DG, we can calculate its degrees of freedom by the same formula we used for calculating the degrees of freedom for a connected component of a constraint network:

$$DOF(DG) = \sum_{o\,in\,DG} DOF_{owned}(o) - \sum_{o\,in\,DG} DOF_{consumed}(o) - \sum_{c\,in\,DG} valency(c) \tag{14}$$

On the other hand, if we know the DOF_{in}'s of all the leaf nodes, we can obtain the same number using a different formula. Suppose that v is the root node of the dependency graph and L is the set of all leaf nodes, then

$$DOF(DG) = DOF_{out}(v) - \sum_{o\,in\,L} DOF_{in}(o) \tag{15}$$

Depending on the sign of the DOF_{in}, we can divide the leaf nodes into three categories (see also Figure 10):

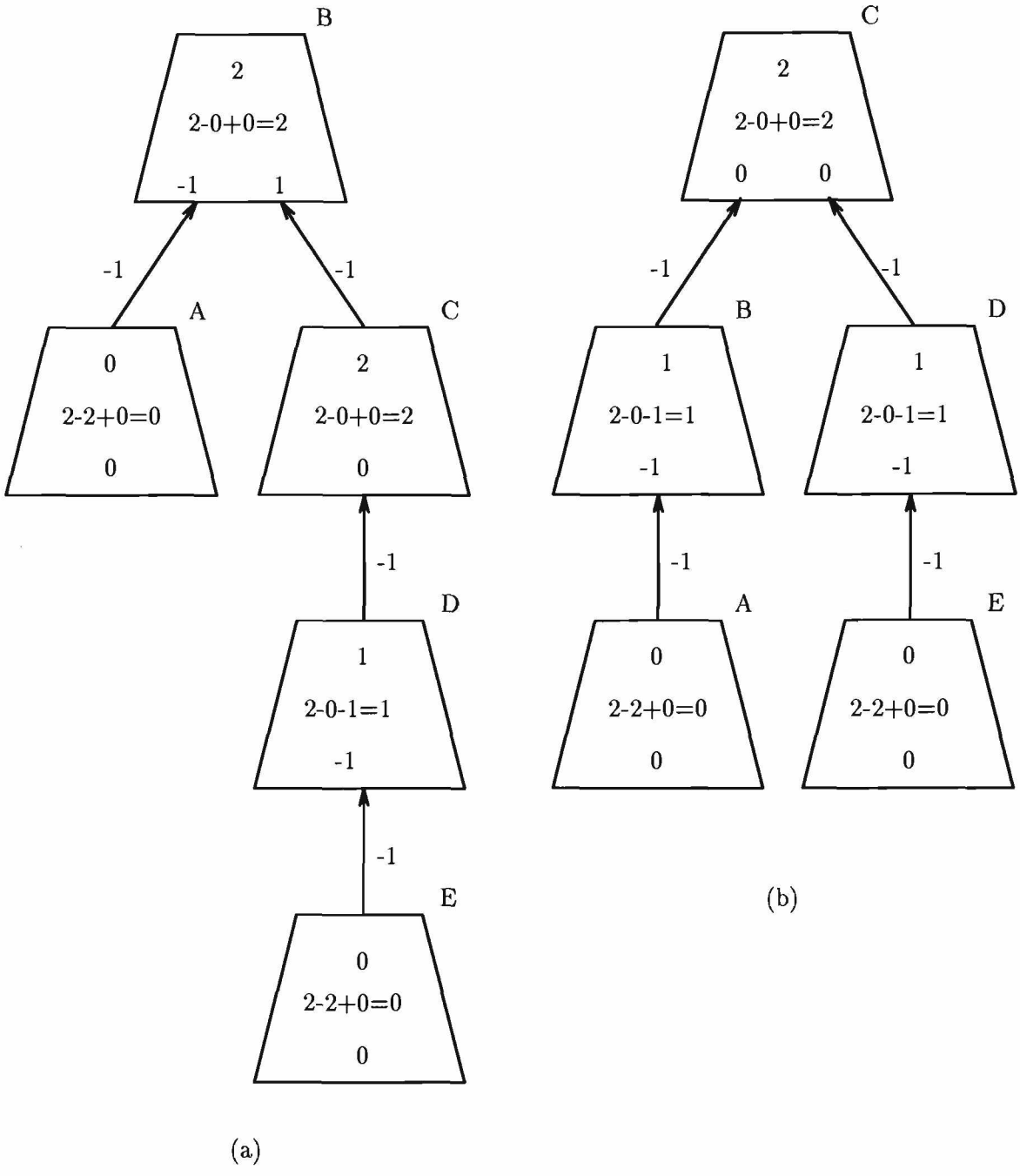


Figure 9: Dependency graphs built under different rules.

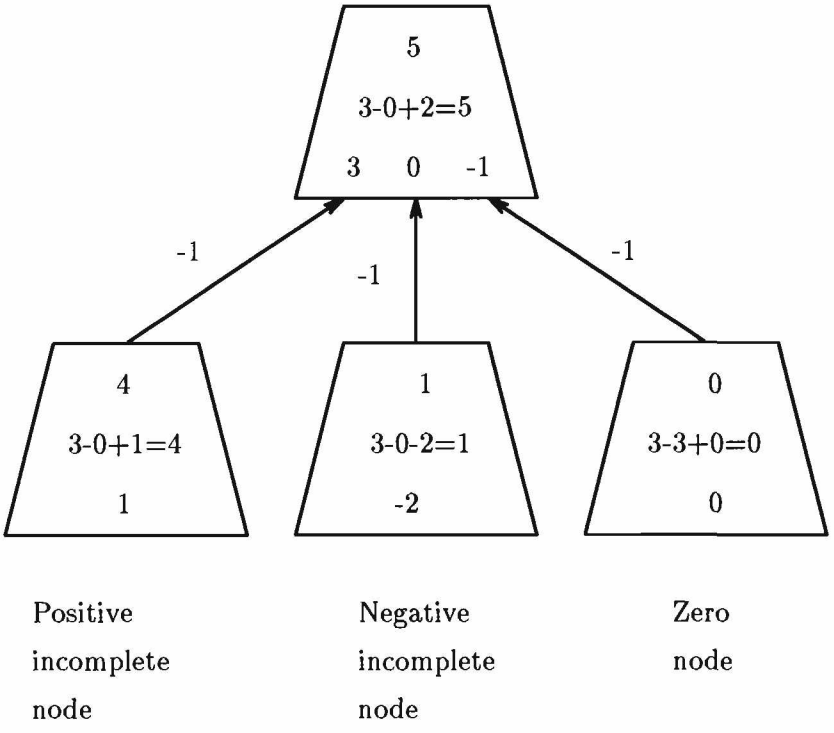


Figure 10: Three different kinds of leaf nodes.

1. A leaf node is called a *zero node* when DOF_{in} is equal to zero. This can be further broken down into two cases: If DOF_{out} is also zero, the node is called a *complete node*, otherwise it is called a *zero incomplete node.*

2. A *positive incomplete node* when DOF_{in} is greater than zero.

3. A *negative incomplete node* when DOF_{in} is less than zero.

If all the leaf nodes are complete nodes, the dependency graph is said to be *complete.* Since all DOF_{in}'s of the leaf nodes are all zero, by equation 15 we come to the following theorem.

Theorem 1 *If the dependency graph DG is complete, then*

$$DOF_{request} = DOF(DG)$$

The reverse is not necessarily true. For example, both of the dependency graphs in Figure 11 satisfy $DOF_{request}$ = DOF(DG), but one of them is not complete because the leaf node is a zero incomplete node.

If exhaustive search is used and the rules are imposed, the leaf nodes will never be positive incomplete.

Theorem 2 *If a dependency graph DG is produced by exhaustive search, then*

$$DOF_{request} \leq DOF(DG)$$

To reveal the actual degrees of freedom obtained, we can *normalize* the dependency graph. To do so, we traverse the dependency graph in a depth-first, bottom-up fashion. Whenever the leaf node is not a zero node, we set it to be zero by adjusting DOF_{in} and DOF_{out}. At a non-leaf node, after we have visited all the incoming arcs, DOF_{in} is summed up again and DOF_{out} is updated accordingly. In the following sections, we will refer to the procedure which traverses a branch bottom-up and rectified the DOF values as *normalization of the branch.*

5 Evaluating the Symbolic Solution

Once we obtain the dependency graph, we will be able to manipulate the corresponding part of the constraint network through the degrees of freedom acquired.

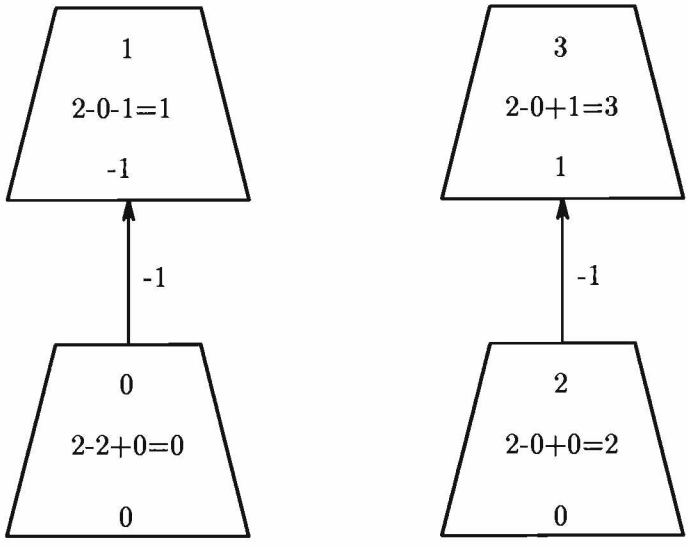


Figure 11: The first graph is complete, but the second is not.

From the theorem above, the dependency graphs produced by the exhaustive search will have some *extra* degrees of freedom, which is 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 and are 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 negative incomplete 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. To be more specific, if two degrees of freedom are requested from a 2-D point, we can manipulating the coordinates of the point by assigning them the coordinates of the locator device.

After nailing 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 any

well 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 B interactively. Since point A is fixed, point B must lie on the circle centered at point A. In other words, it only takes one parameter to completely determine the position of point B. After that is done, the position of point C can be determined by intersecting the two circles centered at point D and new point B respectively. For more detailed information, please refer to [16, 17].

  Another constructive constraint solver quite useful for this purpose is Fudos' bottom-up method [6]. 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 translated 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 B and point C. As before, one degree of freedom of point B need to be nailed interactively. Therefore, 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 structure 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.

6 Variations of the Algorithm

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

6.1 Multi-Rooted Dependency Graph

Suppose we pick multiple nodes from the constraint network and request, possibly different, degrees of freedom from them simultaneously - this situation may occur if, instead of a mouse, we use a pair of data gloves, or other virtual reality devices. With two hands, we are capable of grabbing two objects simultaneously. We can construct the dependency graph in the same breadth-first fashion. As a result, we will get a *multi-rooted dependency graph.* To illustrate this in a simpler way, we introduce *virtual root nodes* for distributing the requested degrees of freedom over the real root nodes. A virtual root node is depicted as a bold trapezoidal box, as shown in Figure 12.

Adding more degrees of freedom to an existing dependency graph can be handled by merging two dependency graphs. We will consider the following cases:

1. If the new root node is not in the existing dependency graph.

   The easiest sub-cases to handle are those when the new dependency graphs do not overlap the existing one, and thus nothing special needed to be done.

   Suppose that the two dependency graphs do overlap. The nodes of the existing dependency graph which may be overlapped are leaf nodes. Moreover, they will be either zero nodes or negative incomplete nodes. Therefore, we need to remove the local constraints imposed by the algorithm previously and to expand them as usual.

2. If the root node is already in the existing dependency graph.

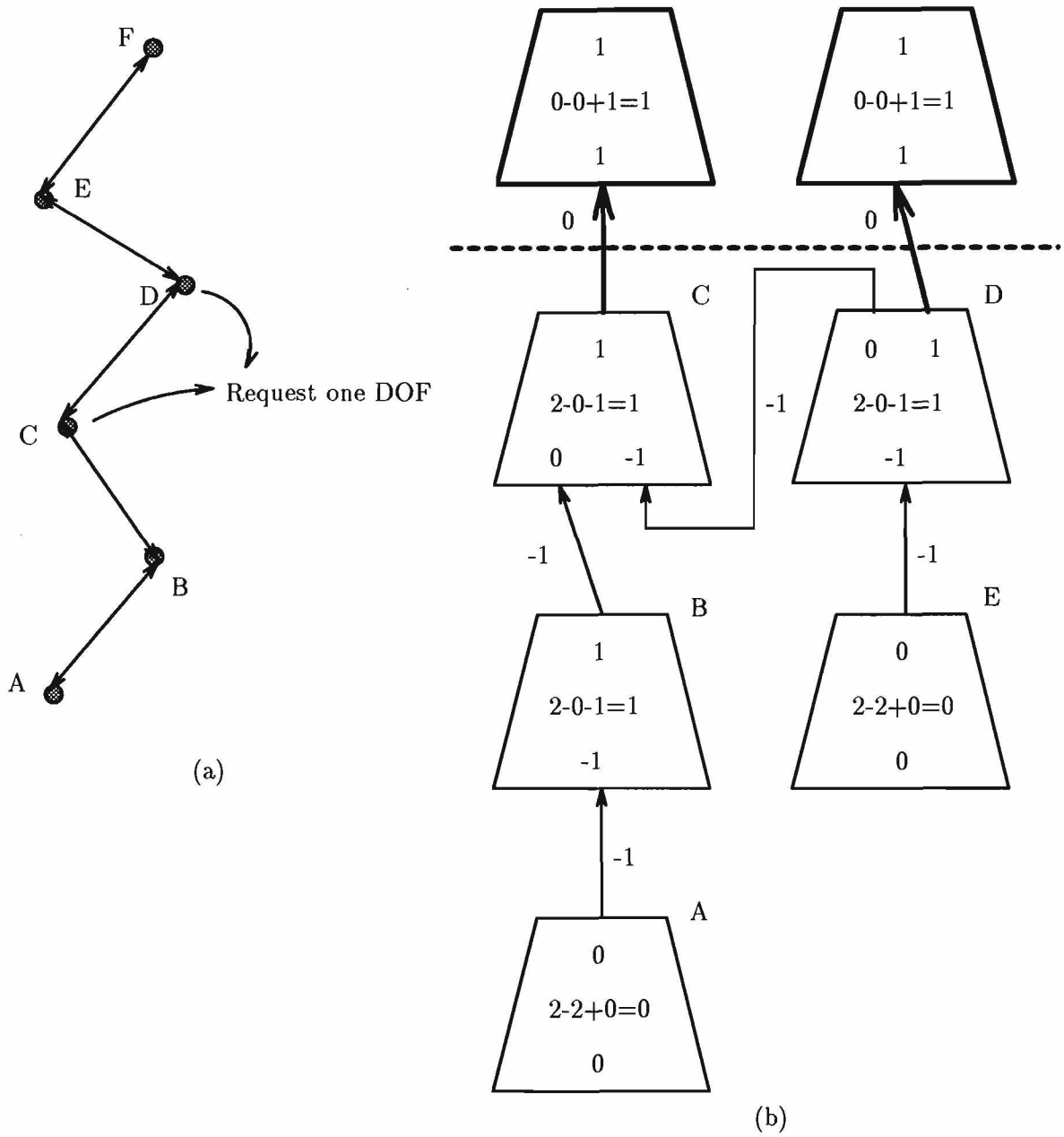


Figure 12: Multi-rooted dependency graph.

In this case, we will need to reconstruct the dependency graph from the point down. We will set DOF_{out} of the node to be DOF_{out} plus the extra degrees of freedom requested. Then this node is expanded with the new DOF_{out}.

We can also retract degrees of freedom from a dependency graph. The maximum degrees of freedom we can retract from a node is equal to the degrees of freedom outputting from that node, i.e. DOF_{out} of the node. This process can be accomplished by reconstructing the subgraph top-down and normalizing the branch bottom-up until a root node is reached.

Valencies can also be added to or subtracted from an arc in the dependency graph. The valencies that can be added to or subtracted from an arc are governed by the basic assumption on the valency of a constraint. For example, the valency of a constraint can not be less than zero or greater than the sum of the degrees of freedom owned by the objects constrained. This process can be also accomplished by reconstructing the branch from the arc top-down and, if the previous construction succeeded, normalizing the branch from the arc bottom-up.

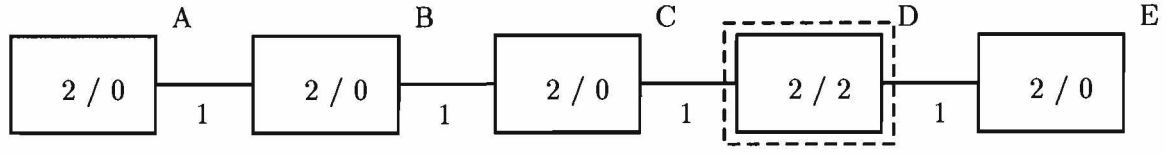
6.2 Controlling the Generation of a Dependency Graph

As we have mentioned earlier, the algorithm for finding a dependency graph is non-deterministic. Therefore, the algorithm will not necessarily produce the intended dependency graph. If we informally refer to the set of all possible dependency graphs for a request as the *solution space* for the algorithm. The basic algorithm just returns the first dependency graph found in its solution space.

A possible way of controlling the search for a dependency graph can be achieved by introducing soft constraints or constraint hierarchies into the system, which will not be discussed here. In this section, we will present a method based on progressively "trimming" the solution space and thus steering the algorithm to the solution we want.

As we have stated before, adding a full local constraint to a node will disconnect the constraint network at that point in terms of passing degrees of freedom. This can be used to trim the solution space. The interactive user may, for instance, temporarily fix some objects, and force the algorithm to backtrack to find alternative dependency graphs.

An example of an altered dependency graph is shown in Figure 13 (c). Figure 13 (b) shows the dependency graph before a temporary full local constraint was applied to point D.



(a)

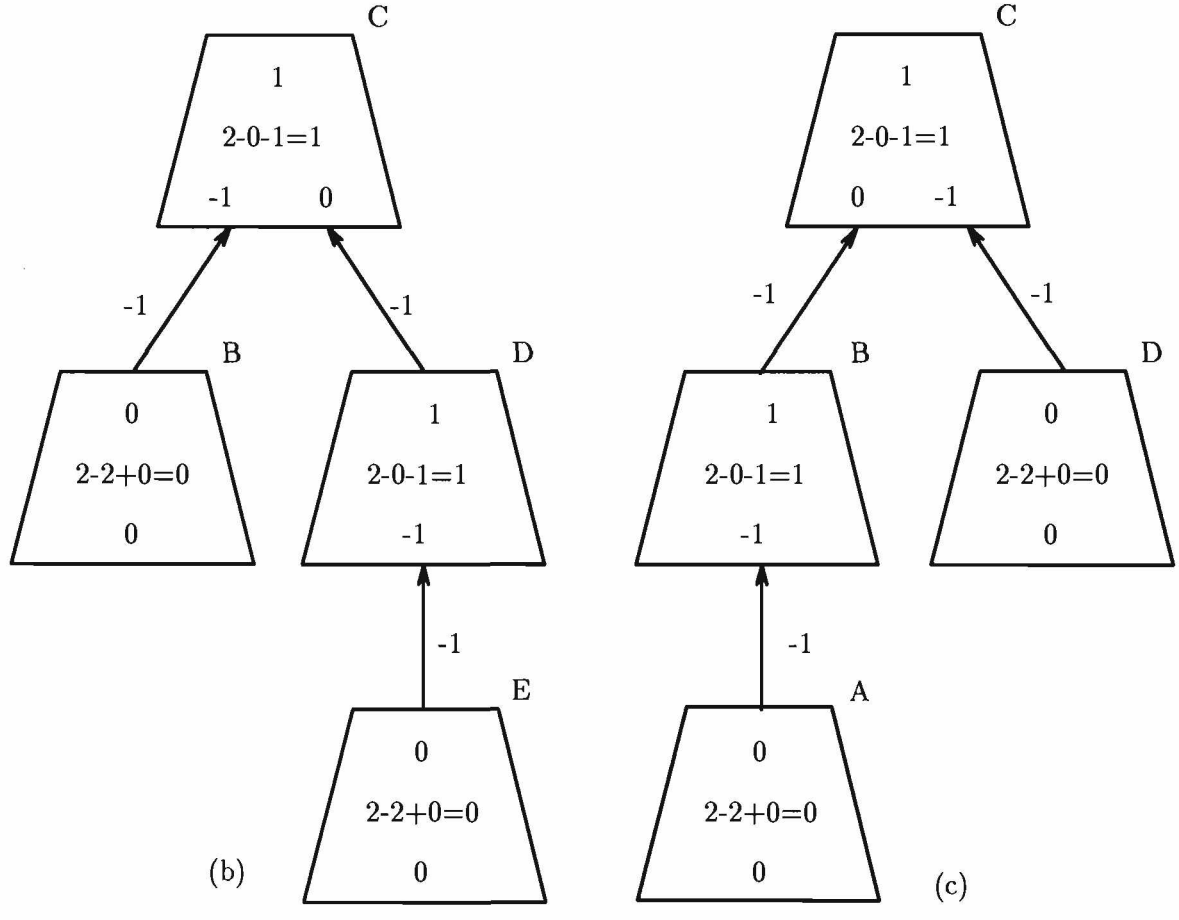


Figure 13: Altering a dependency graph.

One caveat to this approach is that when trimming the solution space, it is possible that the degrees of freedom of the resulting connected component become less than that of requested.

6.3 An Incremental Constraint Solver

Besides maintaining constraint networks, the algorithm can also be used to handle cases where a new constraint is added or the parameter value of an existing constraint is changed. Both of these cases can be dealt with using the same approach, as will be shown.

The first approach is to use the zero degrees of freedom dependency graph as illustrated in section 4.1. Suppose that the parameter of the distance constraint between point A and point B is changed. The dependency graph as shown in figure 7 (a) can be constructed. To re-satisfy the dependency graph, the new position of point B is determined by intersecting the two circles centered at point A and point E respectively. Here the radius of the circle centered at point A is the new parameter value of the distance constraint. Point C is then determined by intersecting the two circles centered at new point B and point D respectively (see also Figure 7 (b)).

A problem of this approach is that the solution is achieved by local changes. It may easily happen that the new solution is outside the range for a certain dependency graph, although it would be feasible for a different dependency graph. This 'range problem' is also experienced during interactive dragging. It can be mitigated somewhat by the user, by interactively changing the position of other points, in order to move the solution into the possible range.

A new constraint can be introduced between objects by simply attaching a new constraint to these objects. The initial value of the constraint is the value determined by the current state of the objects (for instance, if a new distance constraint is defined the initial value is the current distance between these two points). Afterwards the value is updated by means of the above procedure.

If the necessary degrees of freedom cannot be achieved the way it was described above, this means that the new constraint locally over-constrains the specification, and the desired value cannot be attained, in general. Even if the objects already satisfy the new constraint, it constitutes a redundancy, and may be violated later when other values change. Therefore, the new constraint will not be allowed in these cases.

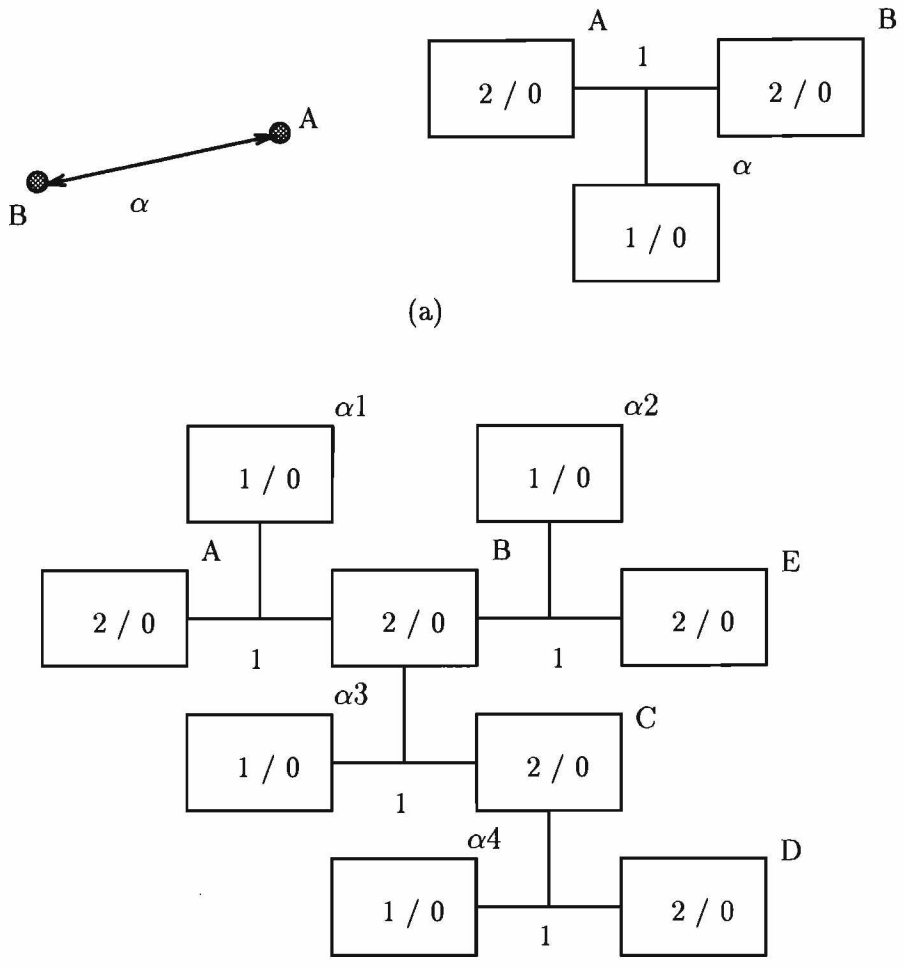


Figure 14: Constraint network with parameters.

6.4 Representing Parameters

Another approach of changing the parameter value of a constraint is to represent the parameter explicitly as an object in the constraint network. Figure 14 (a) shows the representation of a distance constraint with the parameter. Figure 14 (b) shows the same constraint network shown in figure 7 in the new representation.

To change a parameter in the new representation, we only need to make the node representing that parameter the root node, and request one degree of freedom from it. An example dependency graph is shown in figure 15.

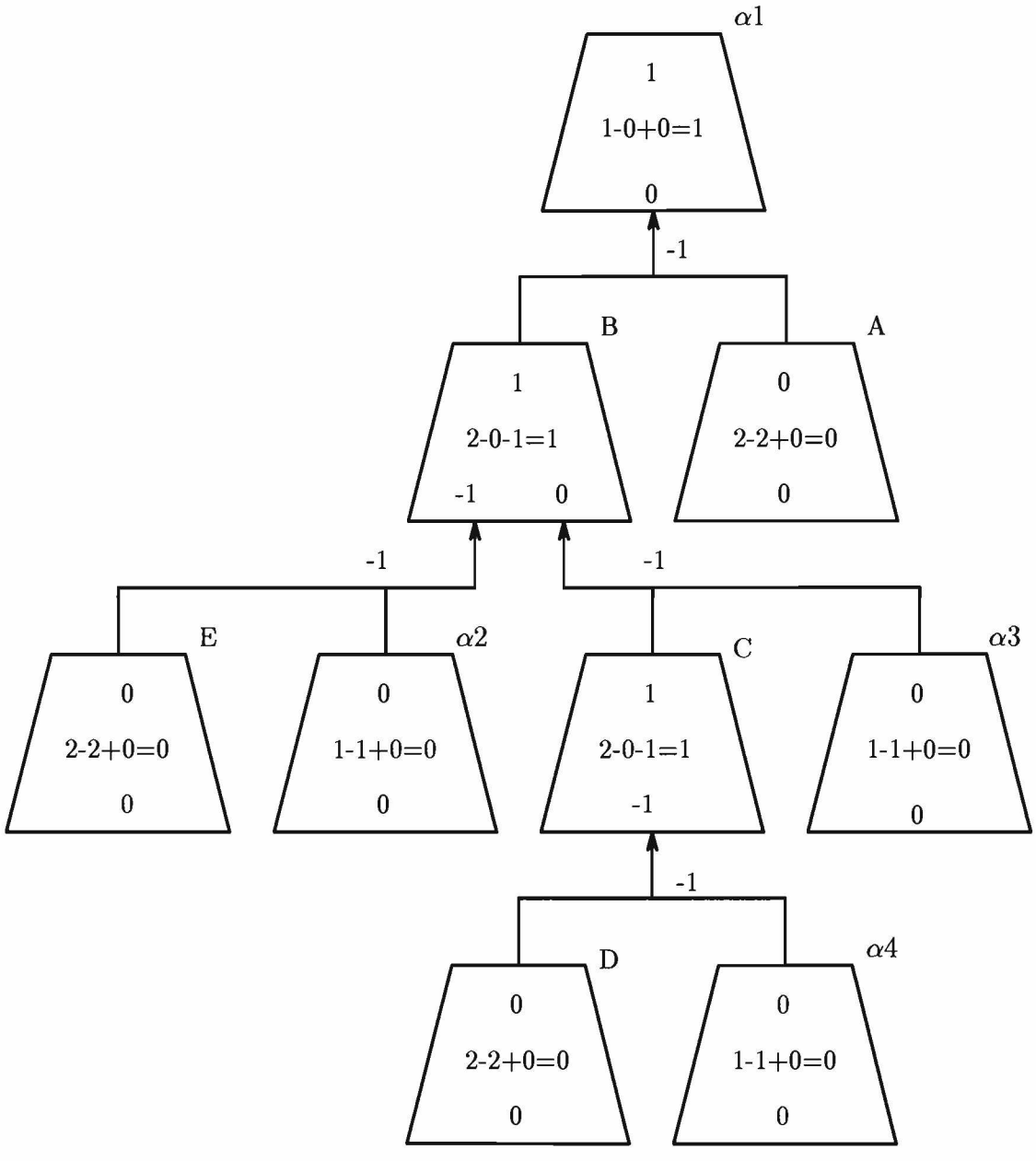


Figure 15: Dependency graph for changing parameter.

Using parameter objects, we can easily 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.

6.5 Algebraic Relations

Using the representation described above, the degrees-of-freedom analysis can also be extended to handle *algebraic relations*, between 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, and show that our algorithm can also be used to derive the undirected graph used by the propagation methods. Take the graph as shown in Figure 16 (a) for example. To change C, a propagation method such as retraction will come up with a plan like the one shown in Figure 16 (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 to use our representation, we have a constraint network as shown in Figure 16 (c). Figure 16 (d) shows a dependency graph equivalent to the plan constructed by the retraction approach.

Next, we will look at an example involving both geometric and algebraic relations. Figure 17 (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 18. 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.
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.

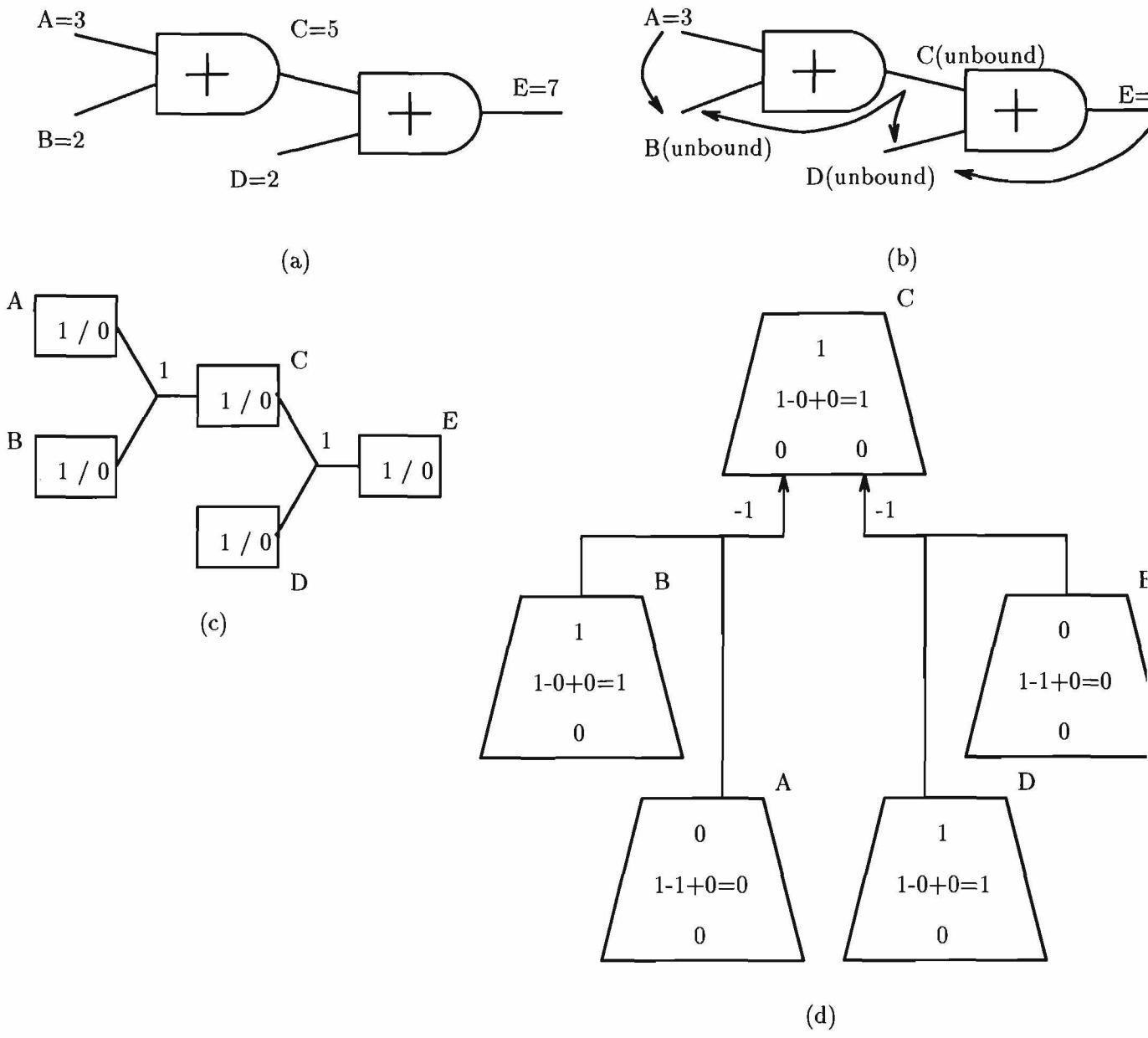


Figure 16: An example of algebraic constraints.

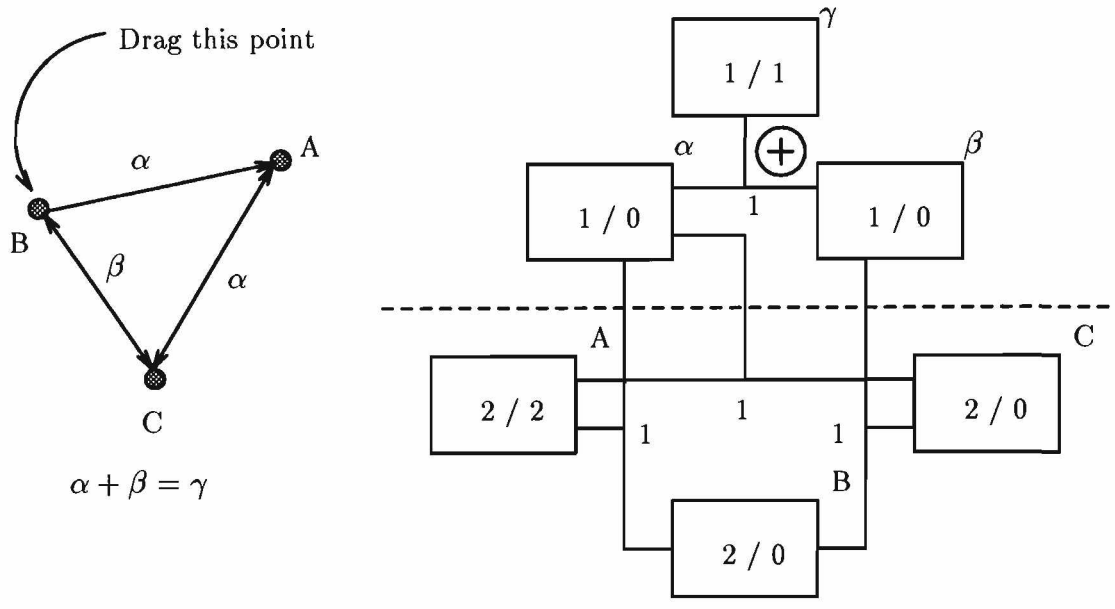


Figure 17: A symmetric triangle with algebraic constraints.

7 A 3-D System

In this section, we will describe a 3-D constraint system which 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 these 3-D models.

7.1 Objects and Constraints

Polyhedra can 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:

$$\vec{p} \cdot \vec{n} = \delta \tag{16}$$

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.

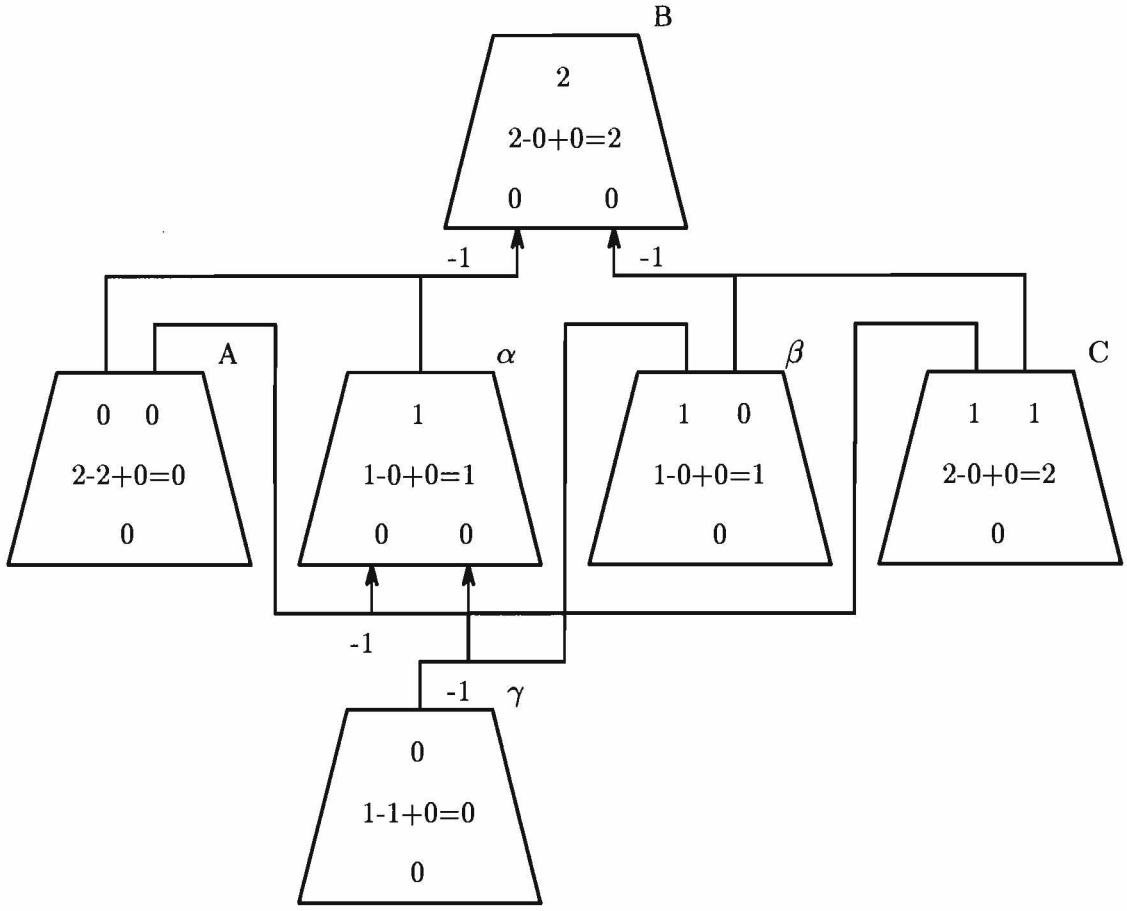


Figure 18: Dependency graph for maintaining algebraic relation as well.

7.1.1 Boundary Representations as Constraint Networks

Boundary representations (B-reps) of polyhedra usually consist of faces, edges, and vertices. 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 19 (a). 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.

A vertex can be derived from three intersecting planes, or by intersecting one edge and one plane, as shown in figure 19 (b) and (c).

Representing a derived object is sometimes useful, for instance, when a constraint is defined on it. There are several types of constraints defined on derived objects. For example, a distance constraints can be defined between a plane and an edge, or between a plane and a vertex, or between two vertices as shown in figure 20.

Figure 21 shows an example polyhedron with notch. 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 22. 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.

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 [28]. One of the resolvable sequences that can be defined for the model above is:

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

In this partial sequene, 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.

The analysis algorithm works well in the above case. However, if we fix vertices v3 and v4 instead, the algorithm will possibly construct a dependency graph similar to the one above (by simply exchanging the labels v3 and v6 in figure 22), but it will fail to evaluate it. The reason is that the algorithm assumes general positions for all objects, and hence it is

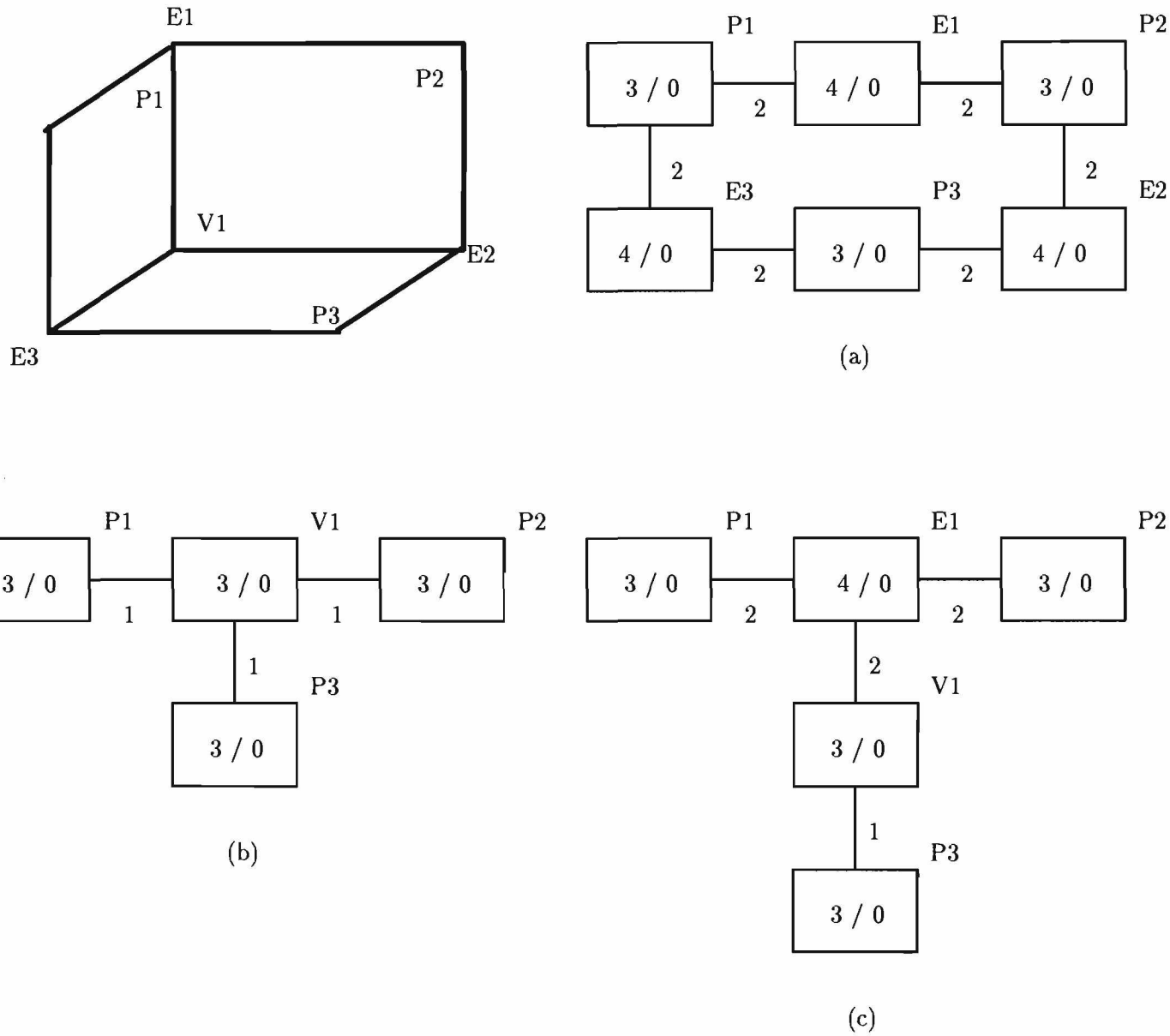
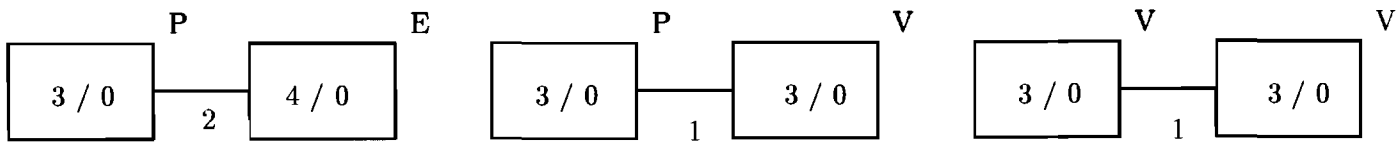


Figure 19: Incidence relation between planes, edges, and vertices.



Distance constraint between a plane and an edge

Distance constraint between a plane and a vertex

Distance constraint between two vertices

Figure 20: Various distance constraint types.

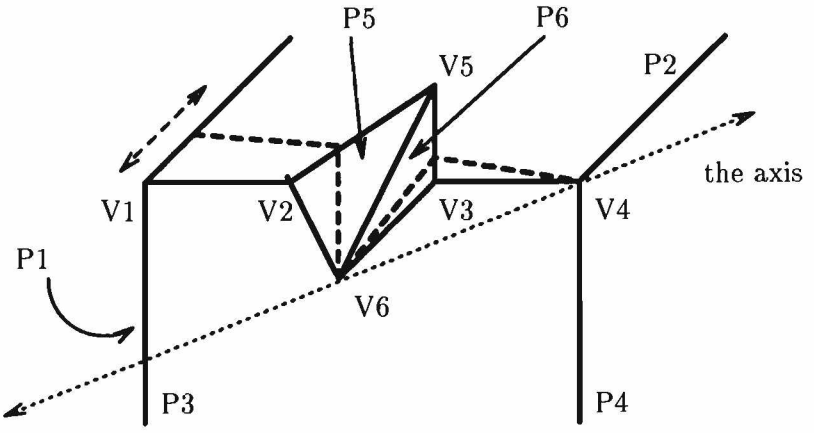
unable to recognize the special case where vertices v1, v3, and v4 are collinear. The proposed solution, namely to rotate p3 about an axis through v3 and v4 does not yield the desired degree of freedom for v1. One remedy is to represent the edges explicitly. This way, the analysis algorithm will realize 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 every edge and vertex 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 23 shows a subset of the constraint network around vertex v1 (points are 9-dimensional, edges are 4-dimensional).

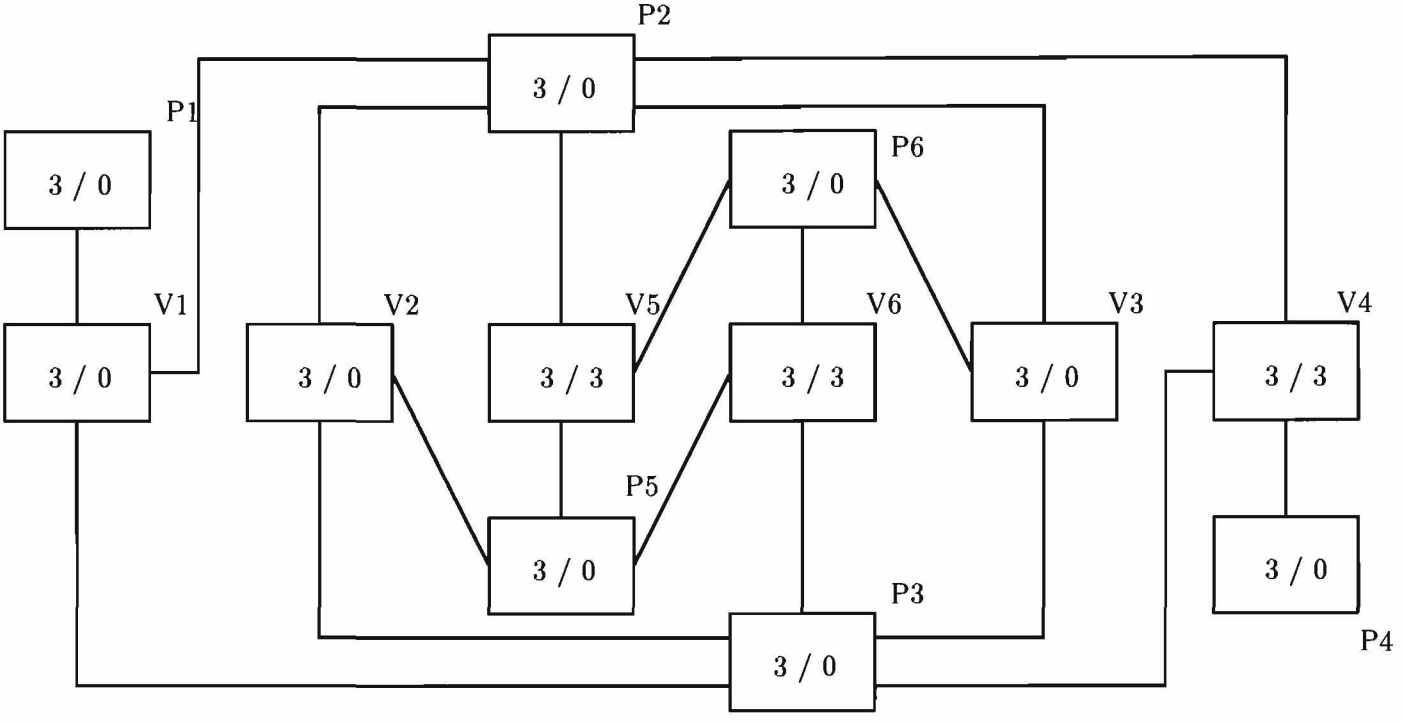
Note that when the planes are manipulated, the position of the edges and the vertices will also change accordingly. Care must be taken of the degenerate cases in the evaluation, where two planes eventually become parallel.

7.1.2 Constraints with Sub-structure

Multi-dimensional objects have different degrees of freedom (e.g. translational, rotational, etc.) which are not necessarily interchangeable, which is illustrated with an example, below.

An angle constraint constrains two intersecting planes to form a specified angle. The valency of the angle constraint is equal to one. The orientation points of the two planes will be a fixed distance apart (depending on the magnitude of the angle). In other words, an angle constraint between two planes maps to a distance constraint between the corresponding





All the constraints have valency equal to one.

Figure 21: A polyhedron with notch.

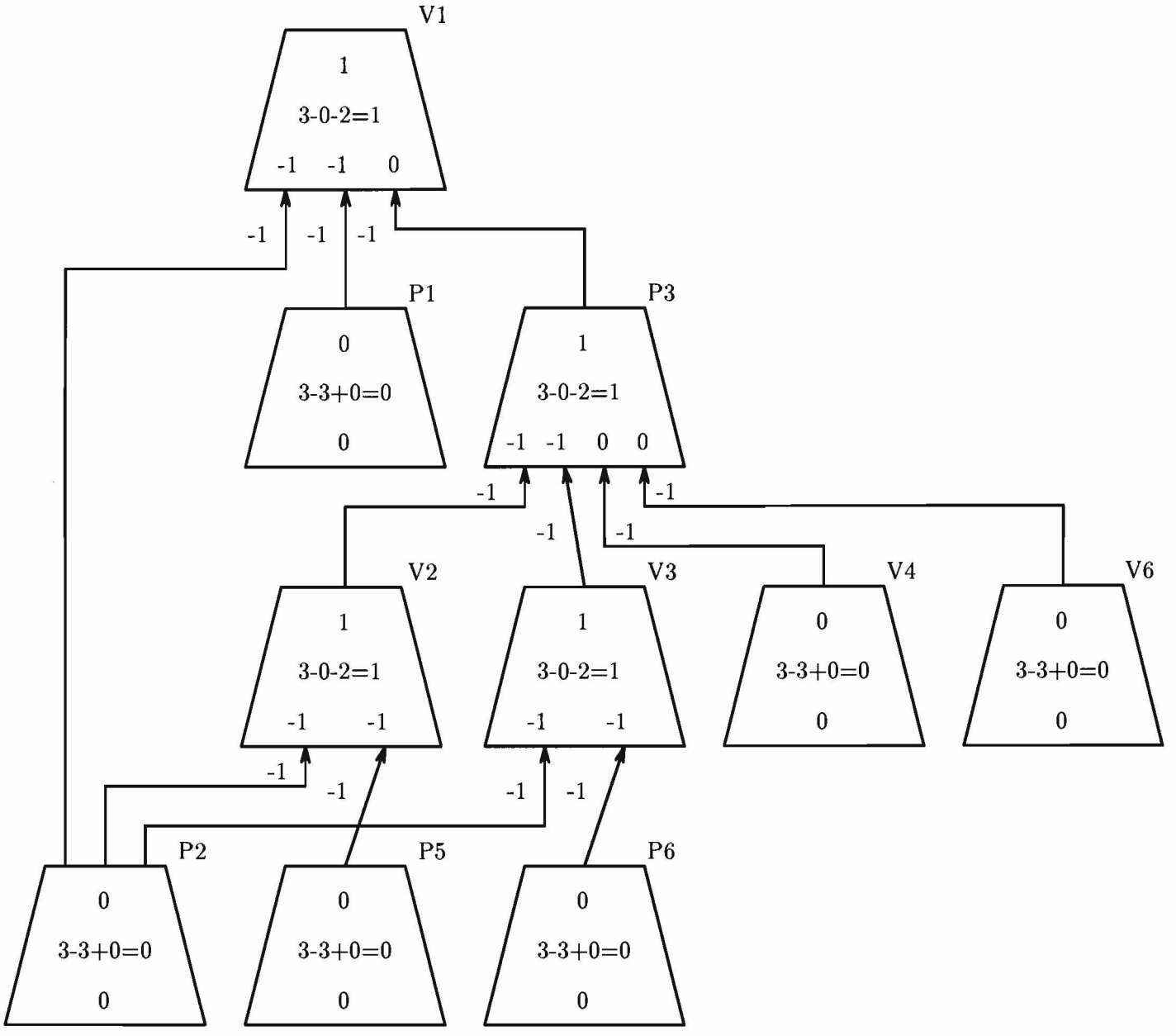


Figure 22: A dependency graph for moving v1 in one degree of freedom.

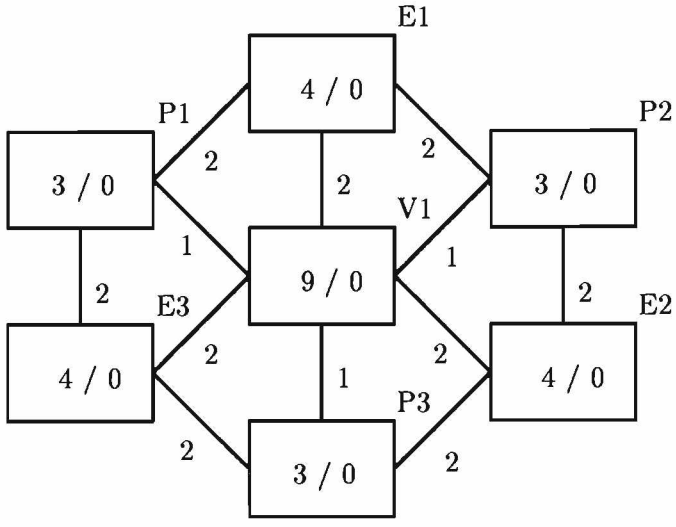


Figure 23: A representation with redundant degrees of freedom and valencies.

orientation points on the surface of the unit sphere. Therefore, if we fix the orientation of one plane, the solution of the orientation point of the other plane will fall on a circle on the unit sphere. If the orientation point of one plane has one degree of freedom, then the other plane will have two degrees of freedom and the solution space forms a circular strip on the unit sphere, in general.

A parallel constraint between two half planes is a special case of the angle constraint (the angle = 0). When two half planes are parallel, their orientation points coincide, and therefore the valency of the constraint is two.

A parallel distance constraint between two half planes has a valency three. It constrains two half planes to be parallel and a fixed distance apart.

An orientation constraint on a plane fixes the orientation of the plane it is a partial local constraint with valency two. This is useful, for instance, when reference planes such as vertical or horizontal ones are employed.

Note that, for example, the rotational and translational degrees of freedom of three parallel planes are not interchangeable. Representing constraints in the usual way (see figure 24) may result in useless dependency graphs, as shown in figure 25. The dependency graph would indicate that we can obtain two degrees of freedom from the first plane, even when fixing the third plane. While it is correct that the second plane has one (translational)

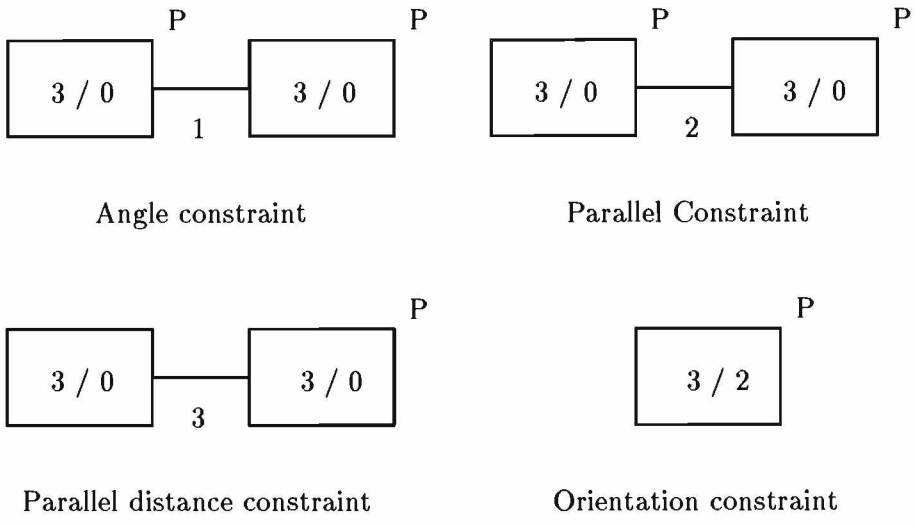


Figure 24: Simple representation of 3-D constraints.

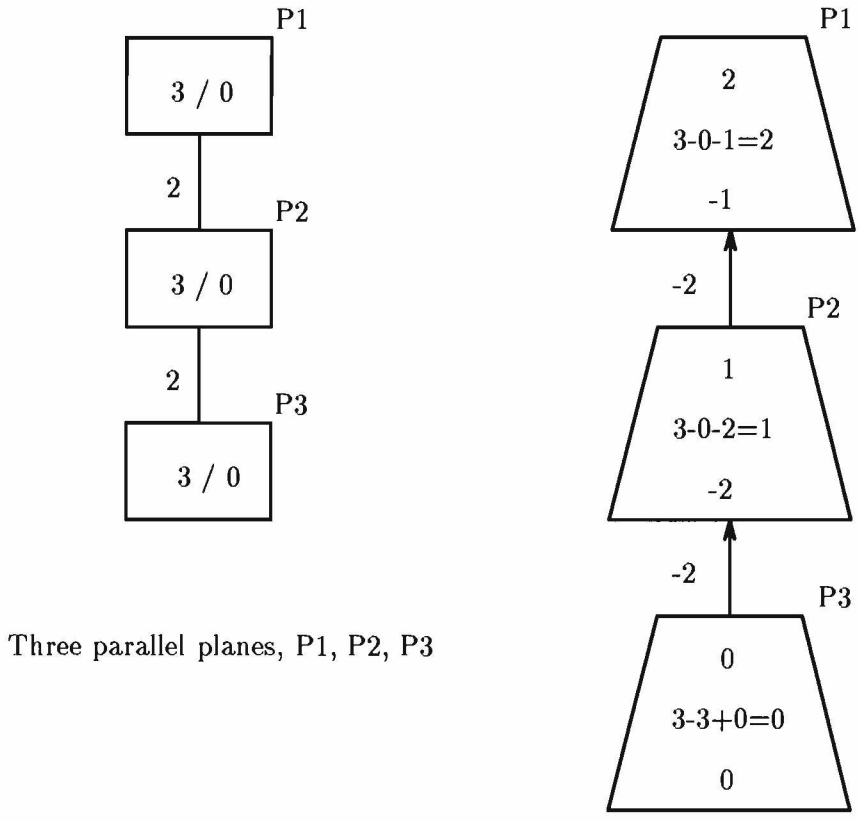


Figure 25: Misleading dependency graph.

degree-of-freedom, and the first plane has one additional degree of freedom, relative to plane 2. However, these two degrees of freedom are identical, which is a degenerate case, and the actual degree of freedom for the first plane is translational and only one dimensional, while the intention possibly was to get two rotational degrees of freedom this way. The problem is that the algorithm assumes that the valency has equal effect on the different degrees of freedom owned by the constrained objects.

The proposed solution is to explicitly represent the different types of degrees of freedom as substructures. The constraints either directly affect these substructures or the object as a whole. Appropriate adjustments need to be made to the analysis algorithm, in that it needs to also decide how to divide the DOF_{out} among the substructures, as illustrated in figure 26 (b). The following balance equations can be applied to objects with substructures: (see also figure 26 (b))

$$\begin{aligned} DOF_{out}(j) &= DOF_{out}(p) + DOF_{out}(q) \\ DOF_{out}(k) &= DOF_{out}(i) + DOF_{out}(q) \\ DOF_{out}(q) &\geq 0 \\ DOF_{out}(k) &\leq \text{the net degrees of freedom in the substructure} \end{aligned} \tag{17}$$

(18)

Solving the above problem of three parallel planes with the new structure yields a correct solution. This time we explicitly request two rotational degrees of freedom, and the dependency graph correctly determines that all three planes therefore have to have two rotational degrees of freedom, and the translational degree of freedom of each plane can be fixed (see Figure 27).

8 Conclusion

The degree of freedom analysis algorithm presented in this paper is a very general tool for reasoning about constraint systems. It works both for geometric constraint systems and algebraic ones.

Also, 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. In addition, it is independent

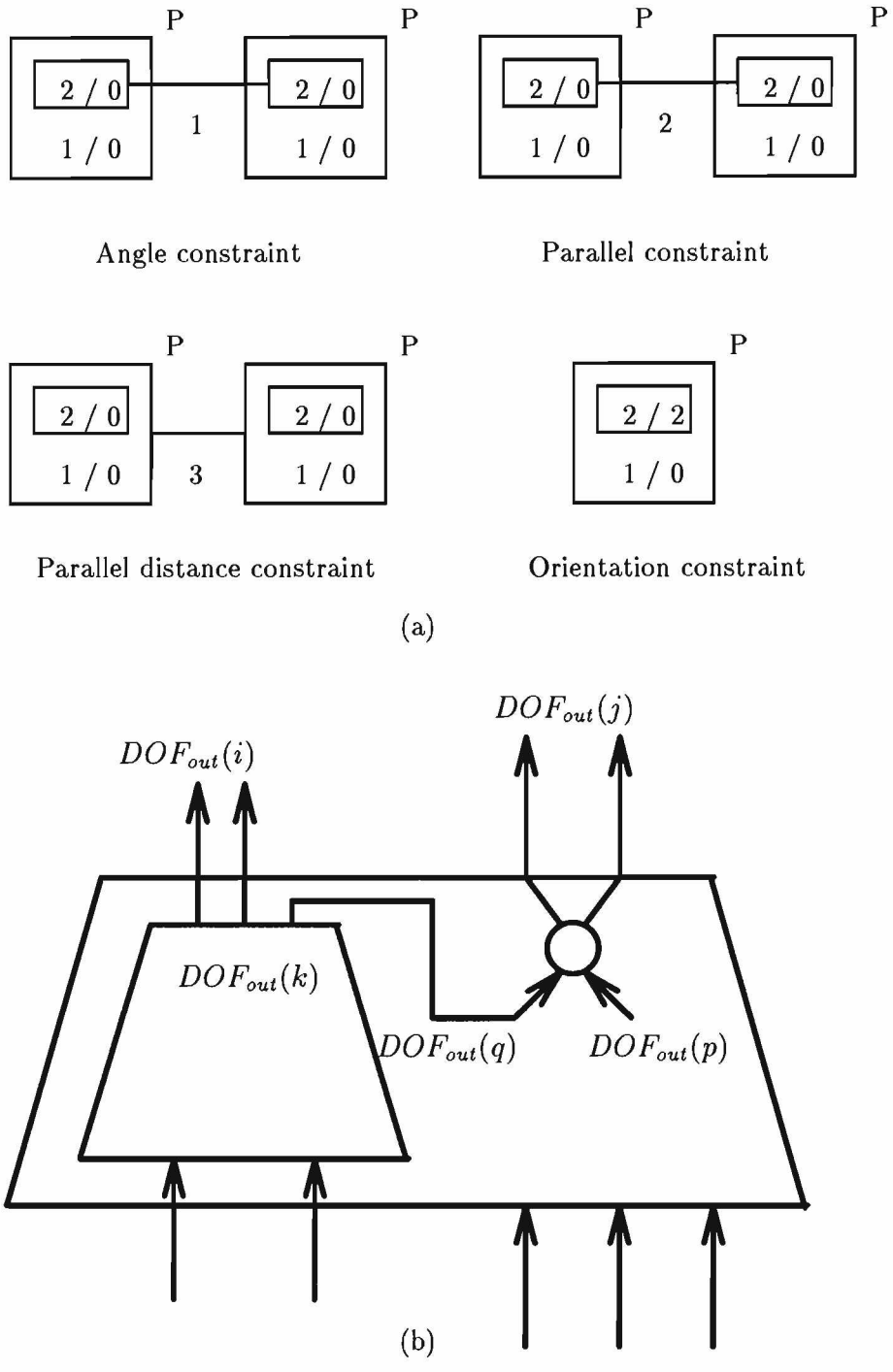
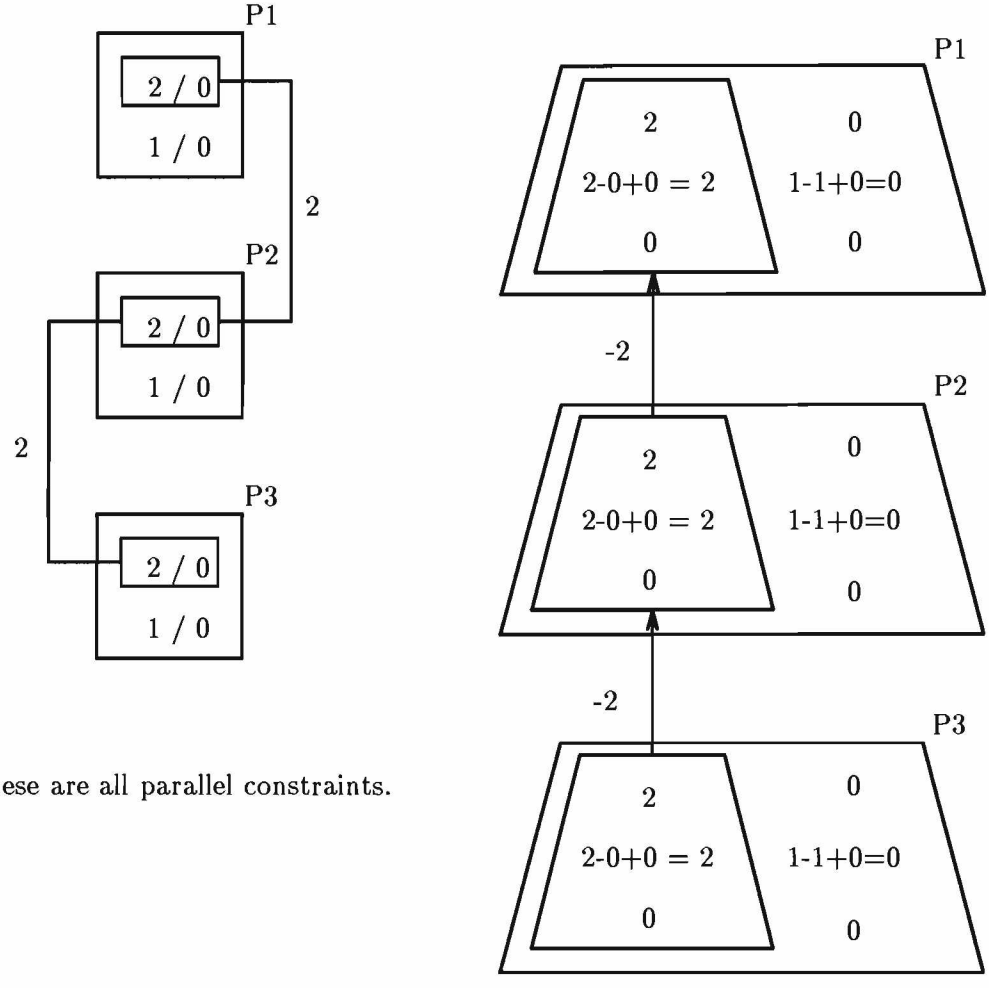


Figure 26: This is the right way to represent the constraints.



These are all parallel constraints.

Figure 27: This dependency graph deduces the right conclusion.

of the evaluation methods used. Almost any constraint evaluation can be used in combination with the algorithm.

The algorithm may prove to be a powerful tool in interactive geometric modeling. Its success, however, will hinge upon powerful evaluation methods, and powerful user interfaces. The framework given here lays the theoretical foundations that will make this possible. 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.

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] ANDERSON, B., AND MOORE, J. *Optimal Filtering.* Prentice-Hall, NJ, 1979.

[4] BARFORD, L. A. *A Graphical, Language-based Editor for Generic Solid Models Represented by Constraints.* PhD thesis, Computer Science Department, Cornell University, May 1987.

[5] 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.

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

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

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

[9] 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.

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

[11] 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.

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

[13] HEL-OR, Y., RAPPOPORT, A., AND WERMAN, M. Relaxed parametric design with probabilistic constraints. In *Proceedings of the 1993 ACM/SIGGRAPH Symposium on Solid Modeling Foundations and CAD/CAM Applications* (Montreal, Canada, May 19-21 1993).

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

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

[16] 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).

[17] 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.

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

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

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

[23] 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).

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

[25] 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.

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

[27] 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.

[28] 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).

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

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

[31] WITKIN, A., FLEISCHER, K., AND BARR, A. Energy constraints on parameterized models. *Computer Graphics 21*, 4 (July 1987), 225–232.