1-1-1990

Constraint-based Programming: A Survey

Jennifer J. Burg

INSTITUTE FOR SIMULATION AND TRAINING

CONSTRAINT-BASED PROGRAMMING:
A Survey

August 31, 1990

IST-TR-90-16

IST

August 31, 1990

Constraint-Based Programming: A Survey

Jennifer J. Burg
Charles E. Hughes
J. Michael Moshell
Sheau-Dong Lang

Institute for Simulation and Training
12424 Research Parkway, Suite 300
Orlando FL 32826

University of Central Florida
Computer Science Department

IST-TR-90-16

CONSTRAINT-BASED PROGRAMMING: A SURVEY

Jennifer J. Burg
Charles E. Hughes
J. Michael Moshell
Sheau-Dong Lang

Computer Science Department &
Institute for Simulation and Training
University of Central Florida
Orlando, FL 32816

8/31/90

Constraint-based programming is an area of computer science research which has been receiving considerable attention in recent years. Intuitively, constraints are understood as limitations or restrictions. Applied to problem-solving and simulation, constraints express the interrelationships among the subparts of a problem or the subsystems of a modeled object. This general foundation in logical or mathematical relations makes constraints applicable in a wide range of fields, including knowledge representation, logic programming, labeling problems, theorem-proving, term rewriting systems, physical modeling in graphics, CAD systems, linear programming, and optimization theory. Our survey discusses constraint-based programming in this variety of contexts.

We divide the survey into finite-domain and continuous-domain constraint satisfaction. Finite domain constraint satisfaction is described as a basic labeling problem to be solved with backtrack-search techniques. Continuous-domain constraint satisfaction defines constraints over real number variables and can involve simplex-like algorithms or algorithms for solving systems of linear *and non-linear equations. In each domain, we review the standard constraint satisfaction methods and existing systems which apply constraints to problem-solving, modeling, and simulation. We conclude with a discussion of constraint logic programming languages which, in one framework, allow constraints to be defined over a variety of domains.*

CONTENTS

INTRODUCTION

Freedom and constraint are two aspects of the same necessity, which is to be what one is and no other.

Antoine de Saint-Exupéry

What is a constraint, and why does this word keep reappearing in computer science literature? The notion of a constraint is an intuitively accessible one. We all-too-frequently bump up against constraints in our everyday lives, and from these encounters we have come to know constraints as limitations, fences in the way of what cannot be. Then what good are they? Perhaps we need to look at constraints from another perspective. St. Exupéry points out that a constraint equally defines what can and must be. It is from this perspective that we begin to see the usefulness of constraints as a modeling and problem-solving tool in computer programming.

A constraint is a relation; one thing is constrained *by* another. Your desire to travel is constrained by your responsibilities at work. The movement of your arm is constrained by its attachment to your shoulder. These are both statements of constraints, but they circumscribe your behavior at different levels. At one level, your actions are determined by your goals and values. At another, your motion is defined by the physical properties of your body. With a constraint description, we can draw boundaries around you, and in doing so we are sketching your outline, focusing the picture more clearly with each constraint.

Constraints are of interest as a programming tool because they provide a natural mode of expression for modeling real and imaginary objects. In computer modeling, the problem is to find an appropriate representation of an object we see or imagine. A constraint-based program describes an object as the sum of its parts and the constraints among them. This approach facilitates the modeling process in that a complex system can be decomposed into its subsystems. Each constraint is in a sense a subsystem, a smaller part of the whole object, a piece which can be more readily grasped and described.

Another attraction of constraints is that they imply more than they say. Your hip bone is connected to your thigh bone, but your thigh bone is connected to your knee bone, and all these connections together affect how you walk. The constraints, or subsystems, interact through their shared parts, and from the parts, we get the whole picture.

This movement from local constraints to global implications makes constraint programming useful in problem-solving. Just as it is easier to describe a complex system in terms of its subsystems, it is often easier to describe a complex problem in terms of subproblems. In systems of simultaneous equations, for example, we may have no trouble expressing in one equation a relation among the variables, but solving the whole system of equations is another matter.

The idea behind constraint-based programming is that a declarative expression of a problem or a declarative description of an object is often more natural than a procedural one. But once the problem is expressed, there must be some mechanism for solving it;

that is, we need a *constraint satisfaction system.* In problem-solving, we can view this system as an algorithm which deduces the global implications of local constraints; that is, it looks at the parts and gives you back the whole. In modeling, we can view it as an overlord which sees to it that an object obeys its own internal laws and the laws of its environment.

In this survey, we will view constraints from a number of perspectives in an effort to understand their usefulness as a modeling and problem-solving tool. We begin by stating more precisely the basic definitions.

1. DEFINITIONS

1.1. Constraint-Based Programming

Constraint-based programming is a declarative style of programming with applications in modeling and simulation, user-interface construction, design and planning, and general problem-solving. A constraint problem requires two parts for its solution:

--a description of the relations between variables and

--a constraint satisfaction system which enforces the relations.

Constraint-based languages stand in contrast to procedural languages, in which a program is a step-by-step specification of how to solve a problem. Constraint programming offers an alternative mode of expression in cases where a declarative description of a problem is more natural than a procedural one. The procedurality is inferred by the constraint satisfaction system, which monitors the variables and enforces the stated constraints.

1.2. Constraints

We define a constraint generally as a relation between or among variables. For example, the statement

$A = 2.3 * B$

is a constraint between the variables A and B. Given a value for B, a constraint satisfaction system should be able to deduce a value for A. Similarly, the system should be able to start from A and derive B. The important point to note is that a constraint is not a procedure; it is an assertion of a fact which must always hold true. Furthermore, a constraint such as this one is bi-directional in that the constraint satisfaction system can propagate values in either direction.

The example above is a numerical constraint which can be satisfied with real number values. A constraint may also involve variables defined over discrete, finite domains. For example, the relation

different_colors (A, B)

might represent a constraint requiring that variables A and B be assigned two different colors from the list *{red, blue, yellow}*.

Although there are some basic similarities between constraints defined over discrete, finite domains and those defined over continuous, numerical domains, they are solved with different techniques and lead to different applications, as discussed below.

2. CONSTRAINT SATISFACTION IN THE FINITE DOMAIN

More than one definition of constraint satisfaction over finite domains has appeared in the literature. The most general formulation is referred to as the consistent labeling problem.

2.1. The Consistent Labeling Problem

The problem of satisfying constraints over a finite domain has been formulated under the name of *the consistent labeling problem* [Haralick and Shapiro 1979]. Haralick and Shapiro identify consistent labeling as a generalization of a number of other well-known problems, including subgraph isomorphism, graph coloring, Boolean satisfiability, and scene labeling.

Haralick and Shapiro define the *compatibility model for the consistent labeling problem* by the quadruple (U,L,T,R), where the following definitions and conditions hold:

$U = \{1,\ldots,M\}$ is a set of *units*.

L is a set of *labels*.

If $u_1,\ldots,u_N \in U$ and $l_1,\ldots,l_N \in L$ then $(l_1,\ldots,l_N)$ is called a *labeling* of $(u_1,\ldots,u_N)$.

$T \subseteq U^N$ is the set of all N-tuples of units which are mutually constrained.

$R \subseteq (U \times L)^N$ is the set of all $2N$-tuples of unit-label pairs $(u_1,l_1,\ldots,u_N,l_N)$, where $(l_1,\ldots,l_N)$ is a *legal labeling* of $(u_1,\ldots,u_N)$.

A labeling $(l_1,\ldots,l_p)$ is a *consistent labeling* of units $(u_1,\ldots,u_p)$

iff $\{i_1,\ldots,i_N\} \subseteq \{1,\ldots,p\}$ and $(u_{i_1},\ldots,u_{i_N}) \in T$ imply $(u_{i_1},l_{i_1},\ldots,u_{i_N},l_{i_N}) \in R$.

The consistent labeling problem is to find all consistent labelings of units *{1,...,M}* with respect to the compatibility model.

More informally, the problem is to assign labels to a set of units. Some of the units are mutually constrained; only certain combinations of labels for these units are compatible. These constraint relations are given in the set *T*, which is a list of the unit-sets which have a constraint among them. The set *R* tells us precisely which combinations of labels are legal for the mutually constrained units.

As an example of a consistent labeling problem, we will use the blocks puzzle described in Figure 1. The task is to fill a rectangular area with a number of blocks. In the consistent labeling formulation, the units to be labeled are the five empty puzzle slots: Positions *1, 2, 3, 4,* and *5*. The possible labels are the six types of blocks: Blocks *A*, *B, C, D, E,* and *F*. Each label can be used any number of times in the labeling. The consistent labeling formulation of the blocks puzzle is as follows:

U = {1,2,3,4,5}
L = {A,B,C,D,E,F}
N = 2, M = 5
T = {(1,2), (2,3), (3,4), (4,5)}
R = { (1,A,2,A), (1,A,2,C), (1,A,2,E), (1,C,2,D), (1,E,2,B), (1,E,2,F),
(2,A,3,A), (2,A,3,C), (2,A,3,E), (2,B,3,D), (2,C,3,D), (2,D,3,A),
(2,D,3,C), (2,D,3,E), (2,E,3,B), (2,E,3,F), (2,F,3,B), (2,F,3,F),
(3,A,4,A), (3,A,4,C), (3,A,4,E), (3,B,4,D), (3,C,4,D), (3,D,4,A),
(3,D,4,C), (3,D,4,E), (3,E,4,B), (3,E,4,F), (3,F,4,B), (3,F,4,F),
(4,A,5,A), (4,B,5,D), (4,C,5,D), (4,D,5,A)}.

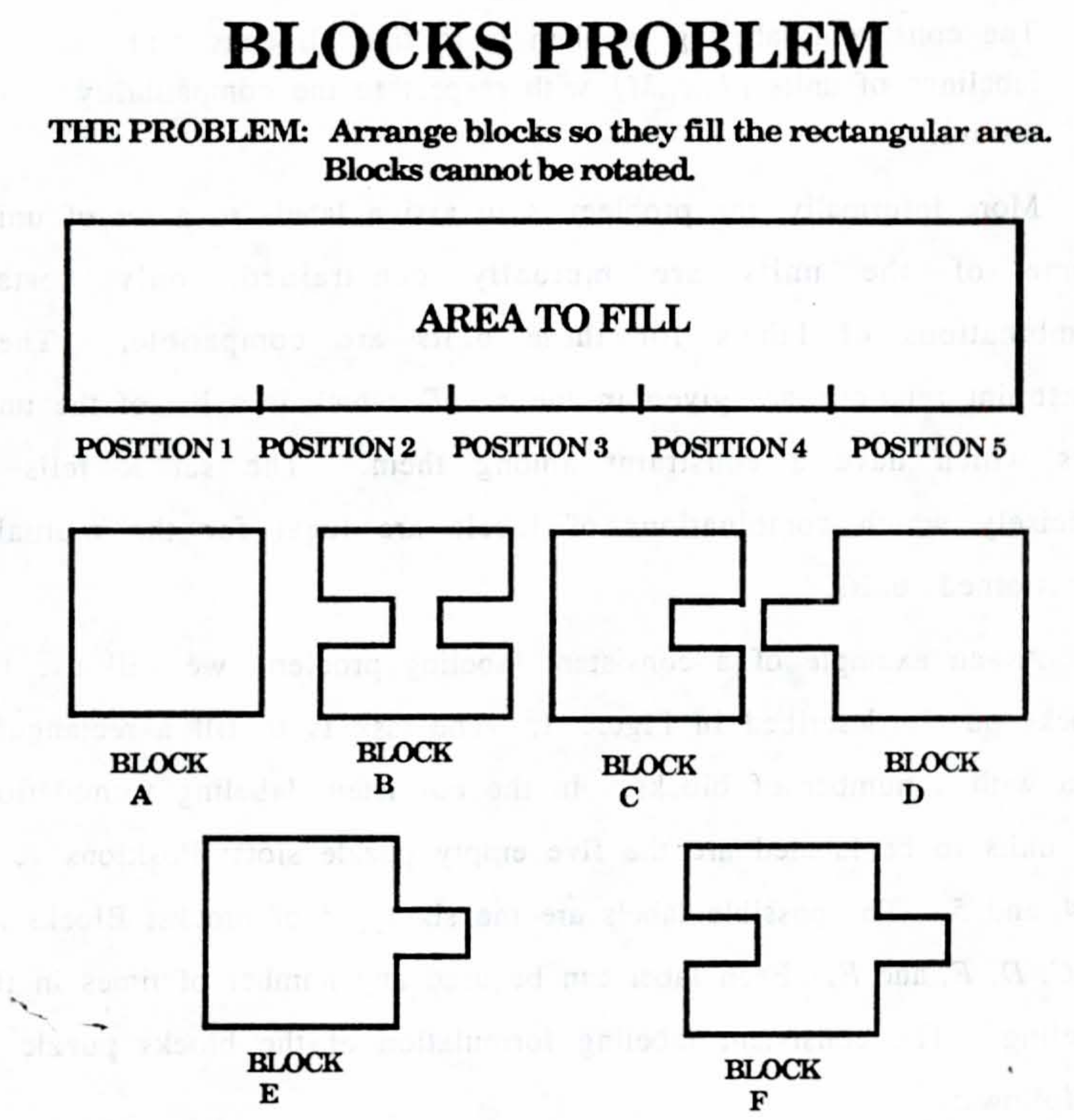


Figure 1. First blocks puzzle example.

T tells which slots are side-by-side and thus are mutually constrained. *R* tells which blocks can fit together in these slots. The tuple *(1,A,2,A)* in *R* indicates that unit *1* can be labeled *A* while *2* is *A*; i.e., a type-*A* block can fit in slot *1* while another type-*A* fits in slot *2*.

Note that if there is no constraint among a subset of units, then there is no restriction in the labeling of these units; all possible combinations of labels are permissible. This lack of constraint is referred to as the *universal constraint.* In the representation of the problem, it is more convenient to record no constraint in T for the universal constraint, and thus no tuples (rather than all possible tuples) are listed in R for the unconstrained units. In our example, blocks which do not touch share the universal constraint.

The problem is to assign labels to all units such that none of the constraints are violated. This involves choosing locally consistent labelings which are also globally consistent.

The basic procedure for solving the consistent labeling problem is a depth-first search with backtracking. First, the units are ordered for labeling. At each node u_i in the search tree, we already have a consistent labeling for units u_1 through u_{i-1} (called the *past units*), and we must find a label for u_i which is consistent with the past units' labels. An assignment of labels to a subset of units is *consistent* if it satisfies all the constraints impinging on those units. A dead-end is reached when no consistent label can be found for u_i, and the search backtracks to try a different label for u_{i-1}. If the dead-end is the root, the search halts with failure to find a solution.

A solution is found when all M units have consistent label assignments. The problem usually entails finding all such solutions, but in some specific constraint problems one solution suffices.

The search tree for general constraint satisfaction is exponential in the worst case, $O(a^n)$ where a is the number of labels and n is the number of units. Clearly, constraint satisfaction is NP-complete since graph coloring, a subproblem of constraint satisfaction, is NP-complete [Haralick et al. 1978].

2.2. Constraint Satisfaction for Binary Constraints

Much of the early work in constraint satisfaction grew out of applications in computer vision, and another definition of the finite-domain constraint satisfaction problem emerged from this work. It was found that scene interpretation problems could be stated conveniently in terms of constraints. (See Section 2.4.5.) Since unary and binary constraints were sufficient for these applications, the definition of the *constraint satisfaction problem* was restricted accordingly, and is as follows[1] [Mackworth 1977; Dechter and Pearl 1988]:

Let $V = \{v_1, v_2, \ldots, v_n\}$ be a set of variables. Let D_i be the domain of possible values for $v_i \in V$. Let $Q_{ij} \subseteq D_i \times D_j$ denote the constraint between v_i and v_j. That is, v_i can have the value x at the same time that v_j has the value y iff $Q_{ij}(x,y)$ is true. A unary constraint on v_i is denoted P_i. That is, variable v_i can have the value x iff $P_i(x)$ is true. Given these variables and the constraints between them, the problem is to find all assignments $\{a_1, a_2, \ldots, a_n\}$ to the respective variables $\{v_1, v_2, \ldots, v_n\}$ such that all constraints hold true.

A matrix is a convenient representation for finite-domain constraint satisfaction. A *1* in position (x,y) of matrix Q_{ij} indicates, for example, that it is permissible for variable v_i to be an x while variable v_j is a y. A *1* in position (x,x) of matrix P_i indicates that it is

[1] It should be noted that not all constraint problems can be stated entirely in terms of unary and binary constraints [Montanari 1974].

permissible for variable v_i to be x. All non-diagonal elements in P_i are 0.

The blocks puzzle is represented as a finite-domain constraint satisfaction problem in Figure 2. A 1 in position (B,D) of Q_{12} indicates that block 1 can be a B while block 2 is a D.

Note that Q_{12} represents which two blocks fit together from left to right, but it does not constrain the first block to one that is flat on its left side. This information is captured in the unary predicate P_1. It is possible to induce a new constraint by performing Boolean matrix multiplication between P_1 and Q_{12} (and similarly between Q_{45} and P_5). The operation $P_1 \cdot Q_{12}$ propagates the implications of the first constraint to the second, and results in a synthesis of the two (Figure 3). (See Section 2.5.3.)

2.3. Constraint Satisfaction and Prolog

A logic programming language such as Prolog can be used as a simple constraint satisfaction system. For example, the blocks puzzle is implemented as a Prolog program in Listing 1.

Let us examine the relationship between finite-domain constraint satisfaction (i.e. consistent labeling) and logic programming more closely. We first need to define some terms. (For good introductory discussions of symbolic logic, logic programming, and mechanical theorem-proving, see [Lloyd 1987] and [Chang and Lee 1973].)

THE BLOCKS PUZZLE AS FINITE DOMAIN CONSTRAINT SATISFACTION

$V = \{v1, v2, v3, v4, v5\}$

$D_1, D_2, D_3, D_4, D_5 = \{A, B, C, D, E, F\}$

	A	B	C	D	E	F
A	1					
B						
C			1			
D						
E					1	
F						

P_1

	A	B	C	D	E	F
A	1					
B		1				
C			1			
D				1		
E					1	
F						1

P_2 , P_3 , P_4

	A	B	C	D	E	F
A	1					
B						
C						
D				1		
E						
F						

P_5

	A	B	C	D	E	F
A	1		1		1	
B				1		
C				1		
D	1		1		1	
E		1				1
F		1				1

$Q_{12}, Q_{23}, Q_{34}, Q_{45}$

Figure 2. Representing a blocks puzzle in the context of constraint satisfaction.

$$\begin{array}{c|cccccc} & A & B & C & D & E & F \\ \hline A & 1 & & 1 & & 1 & \\ B & & & & & & \\ C & & & & 1 & & \\ D & & & & & & \\ E & & 1 & & & & 1 \\ F & & & & & & \end{array} \qquad \begin{array}{c|cccccc} & A & B & C & D & E & F \\ \hline A & 1 & & & & & \\ B & & & & 1 & & \\ C & & & & 1 & & \\ D & 1 & & & & & \\ E & & & & & & \\ F & & & & & & \end{array}$$

$$Q'_{12} = P_1 \bullet Q_{12} \qquad Q'_{45} = Q_{45} \bullet P_5$$

Figure 3. Synthesizing new constraints by matrix operations on known constraints.

Prolog is based upon predicate calculus, or more precisely, Horn clause logic [Clocksin and Mellish 1987]. In Prolog, an object is represented by a *term*. *Constants* are terms which refer to specific objects, while *variables* are terms which refer to different objects at different times. (Constants correspond to labels and variables correspond to units in the consistent labeling problem.) A term can also take the form $f(t_1,...,t_n)$, where f is an n-ary function and $t_1,...,t_n$ are terms.

In order to reason about objects, we express propositions about them. An *atomic proposition* (or, more simply, an *atom*) consists of a *predicate symbol* followed by an ordered sequence of terms which are its arguments. An atom or its negation is called a *literal*.

Propositions express relations (i.e. constraints) among objects. The predicate symbol gives a name to the relation. For example, *neighbors(V1,V2)* is used to represent the fact that the first two positions in the puzzle are constrained to be neighboring blocks. The predicate is true when the constraint is satisfied.

```
blocks_puzzle(V1,V2,V3,V4,V5) :-
      leftend(V1),
      neighbors(V1,V2),
      neighbors(V2,V3),
      neighbors(V3,V4),
      neighbors(V4,V5),
      rightend(V5).
leftend(a).
leftend(c).
leftend(e).
neighbors(a,a).
neighbors(a,c).
neighbors(a,e).
neighbors(b,d).
neighbors(c,d).
neighbors(d,a).
neighbors(d,c).
neighbors(d,e).
neighbors(e,b).
neighbors(e,f).
neighbors(f,b).
neighbors(f,f).
rightend(a).
rightend(d).
```

Listing 1. Prolog program for blocks problem

In predicate calculus, a *clause* is defined as a finite disjunction of zero or more literals (i.e., literals connected by *ors*). The proposition

$\forall x_1 \ldots \forall x_s (p_1 \vee \ldots \vee p_m)$,

is a clause, where each p_i is a literal and $x_1, \ldots, x_s$ are all the variables occurring in $p_1 \vee \ldots \vee p_m$.

Statements in a Prolog program are also referred to as *clauses*, although superficially they take a different form from the proposition above. These clauses have two possible forms. A *unit clause* contains only one literal. Compound clauses take the form

$q \text{ :- } p_1, p_2, \ldots, p_n,$

where q and $p_1, p_2, \ldots, p_n$ are literals. q is the head of the clause; $p_1, p_2, \ldots, p_n$ is the body. (A unit clause has a head but no body.)

In predicate calculus, the above Prolog clause would be written as

$p_1 \wedge p_2 \wedge \ldots \wedge p_n \rightarrow q.$

Note that this clause is logically equivalent to

$\neg p_1 \vee \neg p_2 \vee \ldots \vee \neg p_n \vee q.$

Thus, all clauses in Prolog are *Horn clauses*, i.e., clauses with at most one unnegated literal. Restriction of Prolog to Horn clause logic simplifies the execution of a program.

The first clause in the Prolog program above states that the *blocks_puzzle* relation is satisfied if the *leftend*, *rightend*, and all the *neighbors* relations are satisfied. *Ground unit clauses* (i.e. those containing constants rather than variables) correspond to the tuples which specify the constraints in the consistent labeling problem. They tell precisely which objects fulfill each relation. (Here we see a difference in the consistent labeling problem and a Prolog program: Since a predicate used in the body of a clause can also be used in the head of another clause, we get a kind of part-subpart decomposition of the problem in Prolog. In the consistent labeling problem, one constraint cannot be defined in terms of another.)

A *goal clause* in Prolog is a clause of the form

$?\text{-}\ r_1,\ldots,r_n.$

Each r_i is a subgoal of the clause. A goal clause can also be called a *query*.

Although logic programming is intended to be essentially declarative, the logic of Prolog has a procedural interpretation which is easily understood. A program clause

$q \text{ :- } p_1,\ldots,p_n$

can be viewed as a procedure definition. A program begins with an initial goal. Say we are given a goal clause

$?\text{-}\ r_1,\ldots,r_n.$

Each r_i can be thought of as a procedure call. If the current goal is

$?\text{-}\ r_1,\ldots,r_n,$

a step in the computation entails matching r_1 with the head q of some program clause $q \text{ :- } p_1,\ldots,p_n.$ This matching process involves a consistent substitution of terms for variables, which substitution *unifies* r_1 and q (makes them the same expression.) Once this occurs, the current goal becomes:

$?\text{-}\ (p_1,\ldots,p_n,r_2\ldots,r_n)\theta,$

where θ is a postfix function that represents the unifying substitution. From a procedural point of view, substitution can be viewed as the equivalent of parameter passing.

Since unit clauses have no body, they do not cause any more literals to be added to the goal. Thus, all literals are eventually "erased," and computation terminates when an empty goal is reached.

In our example, the goal is:

?- blocks_puzzle (A,B,C,D,E).

As execution begins, the goal must be matched with one of the clauses in the program. To match, the literals must have the same predicate symbols and the same number of terms as their arguments. Constants must be matched exactly. A variable can be matched to a constant, and the match causes the variable to be bound to the value of the constant. In our simple constraint satisfaction program above, variables receive values which satisfy the constraints when a match is made with a ground unit clause.

A Prolog program can also be viewed as a resolution theorem-proving system. Based upon a set of axioms (i.e. the clauses) which express what we know about our "world," we would like to see what can be logically inferred. The inference mechanism which controls the execution of a Prolog program is based upon the resolution principle [Robinson 1965]. When we issue the goal $?\text{-}\ r_1,\ldots,r_n$, what we really wish to prove is

$\exists x_1 \ldots \exists x_r (r_1 \wedge \ldots \wedge r_n)$.

The goal is in fact the negation of the above statement. Resolution theorem provers work by refutation. That is, if the negation of the proposition to be proved is added to the axioms, and if a contradiction is logically derived, then the proposition is true. Thus, in Prolog, if we can finally arrive at an empty goal, we have shown that there exist values for our variables such that all the propositions in the goal are true. During execution, these values have been bound to the variables. If there is more than one set of values, Prolog can produce them all.

Prolog's unification algorithm gives it the power to handle simple finite-domain constraint satisfaction problems. Like the general constraint satisfaction algorithm described in Section 2.1, Prolog's inference mechanism uses matching and backtrack-search techniques. However, we will see in Section 4 that Prolog has only limited ability in the domain of numerical constraint satisfaction. Prolog II [Colmerauer 1986] was a first step toward replacing the unification algorithm with a more general strategy founded upon constraint satisfaction. Prolog III [Colmerauer 1990] extends this work by providing for the definition of constraints over Booleans, integers, rational numbers, and lists, with the corresponding operations in these domains. Prolog III is discussed in Section 4.1.

2.4. Finite-Domain Constraint Satisfaction Systems

Because of its similarity to logic programming, theorem-proving systems [Fikes and Nilsson 1971; Loveland 1978], truth maintenance systems [Doyle 1979], and general problem-solving [Newell and Simon 1972], finite-domain constraint satisfaction is generally classed as an artificial intelligence problem. Constraint satisfaction is also related to Minsky's frame representation language [Minsky 1975] and object-oriented programming because constraints are a matter of how one chooses to represent knowledge in the modeling process -- i.e., declaratively, in a part-subpart decomposition, with encapsulation of the modeled object.

The movement from local relations within an object to the implied relation within the total object makes finite-domain constraint satisfaction applicable to design, modeling, and planning problems.

2.4.1. MOLGEN

MOLGEN [Stefik 1981] is a knowledge-based hierarchical planner which takes a description of a hormone to be synthesized in a molecular genetics experiment and formulates a plan for synthesizing it. Like the General Problem Solver [Newell and Simon 1972], MOLGEN compares goals, finds differences, and chooses operators to reduce differences. The planner operates upon a number of plan variables representing laboratory objects, placing constraints between objects dynamically to ensure that they are compatible for a given experiment.

The use of constraints in MOLGEN is motivated by Stefik's view of the design and modeling process, which he sees as the decomposition of a complex system into subsystems. He notes that the variables shared by two different constraints provide a channel of communication between two different subsystems of an object. Similar to designing and modeling, planning is a problem suitably represented by constraints, in that constraints can express a partial description or commitment which will be refined during planning.

2.4.2. GARI

GARI is another knowledge-based system for process planning, in this case applied to the metal-cutting industry [Descotte and Latombe 1985]. Given a description of a metal part, GARI produces a plan of cuts to be executed, the order of execution, the machine tools needed, and so on. Knowledge is contained in production rules of the form *conditions → pieces of advice.* The left-hand side consists of conditions about the part, the machines, or the plan itself. The right-hand side constitutes the constraints. The constraints tell the system more efficient or careful ways of making cuts. Thus GARI applies constraints differently from MOLGEN, using them to represent preferences or pieces of advice. (ThingLab, discussed in Section 3.3.2.2, also gives weights to constraints.)

2.4.3. User Interfaces for CAD Systems

Grossman and Ege [1987] use constraints in a floor-layout problem. The idea is to allow more than one designer to work on a design simultaneously from different workstations. For example, an architect can design a floor in a house by inserting walls, doors, and windows. An interior designer at the same time furnishes the room with desks, chairs, and sofas. A logic programming implementation helps to maintain consistency between one worker's view and another's. Objects in the room are represented by logical variables which are instantiated in the design process. In this application, constraints serve as elimination rules since the binding of, for example, a wall variable may eliminate some choices which the interior designer might have otherwise made in laying out the furniture.

2.4.4. PROTEAN

PROTEAN is a knowledge-based system for identifying the three-dimensional conformations of proteins in solution on the basis of empirically-derived constraints [Hayes-Roth et al. 1986]. The constraints reflect what is known about the architecture of helices, covalent bonds, amino acids, etc. PROTEAN differs from the applications above in that it attempts to model a real-world object about which the modeler has only partial knowledge, represented in the constraints placed on the object.

2.4.5. Scene Interpretation

Scene interpretation problems are a natural application of constraint satisfaction since they begin with local information about pixels, edges or regions and move to a more global interpretation. Junction labeling is one specific vision problem represented well by constraints. Edge detection algorithms first analyze a scene in terms of 2-dimensional lines meeting at junctions. Once edges are identified, the next step is to label junctions such that their connections are physically possible. Here we find a constraint problem: Neighboring junctions are constrained in that the edges connecting them must have the same interpretation at each end. For example, if an edge in one junction is labeled an occluding edge, it must be identified consistently as occluding in any other junction which shares that edge.

Constraint-based formulations of low-level vision problems are surveyed in [Davis and Rosenfeld 1981].

2.5. Reducing the Search Space

2.5.1. Waltz Filtering and Node, Arc, Path Consistency

Scene interpretation was the first application of constraints to be of real practical use. The general problem of constraint satisfaction is intractable in the worst case [Haralick et al. 1978]. One way to deal with the computational complexity of constraint satisfaction is to devise an algorithm which falls short of achieving a full answer, but which stills yields some useful information. Waltz filtering [Waltz 1975], applied to junction labeling, is such an algorithm.

Waltz recognized that it is a simple matter to eliminate many of the dead-end branches of the search tree. We can do so by pruning the domains of the variables, eliminating values which we know cannot contribute to a solution. For example, if there exists a value x in the domain of v_i which does not satisfy a unary constraint P_i placed on v_i, then x cannot be used in a solution and can be thrown out of the domain.

This type of inconsistency is called node inconsistency, the term arising from a graphical depiction of a constraint problem. A constraint satisfaction problem restricted to binary constraints can be represented as a *constraint graph* (or *network*), where the variables to be labeled constitute the nodes of the graph, and a constraint between two variables is represented by an arc. Implicit in each arc is a set of variable-value pairs specifying permissible local labelings. No arc is shown in the graph for variables which have no constraints between them; conversely, when there is no arc between nodes v_i and v_j, we can assume that $Q_{ij} = D_i \times D_j$.

A node v_i in a constraint graph is said to be *node consistent* iff for each value $x \in D_i$, $P_i(x)$ is true.

Waltz noted another type of inconsistency in the following situation: Consider a variable v_i given a label x. If for some other variable v_j, there is no value available which is compatible with v_i's value of x, then v_i cannot possibly be labeled x, and we can eliminate x from v_i's domain. This observation led to the definition of arc consistency:

An arc (v_i,v_j) in a constraint graph is *arc consistent* iff for each value $x \in D_i$, there is a value $y \in D_j$ such that $Q_{ij}(x,y)$ is true [Mackworth 1977].

It is easy to see how this pruning of domains for arc consistency could in turn reduce the search. If v_j has no value compatible with the value of x for v_i, then the presence of x in v_i's domain could lead to thrashing in the backtrack search. An example of this thrashing is illustrated in Figure 4.

We are looking at the piece of the tree where v_i received the label x. There is another variable v_j, to be labeled after v_i, which has no value compatible with $v_i = x$. The variables between v_i and v_j receive their values. The problem is that each time the search gets to v_j, it has reached a dead-end. The procedure will continually relabel v_{j-1} (and also v_{j-2} back to v_{i+1}), each time searching through all of v_j's values without success.

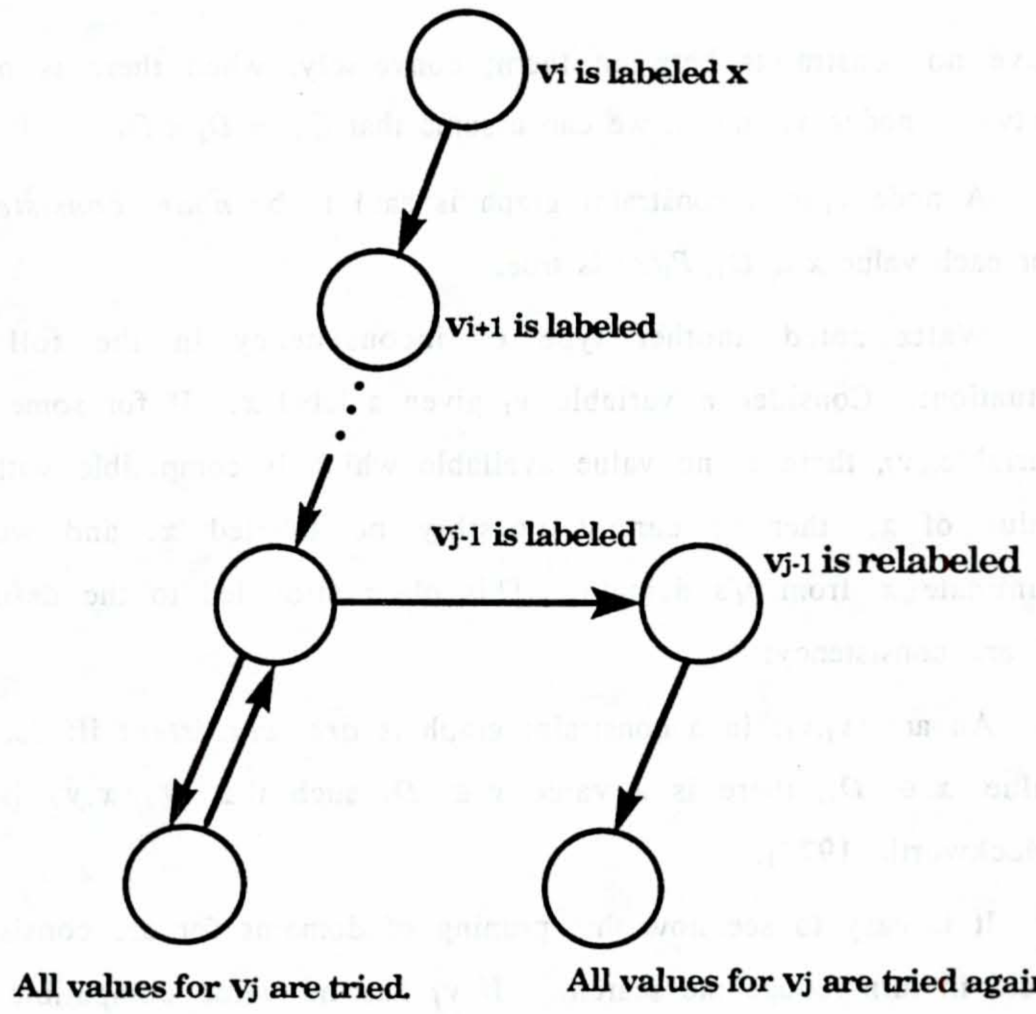

Vi is labeled x
Vi+1 is labeled
Vj-1 is labeled
Vj-1 is relabeled
All values for Vj are tried.
All values for Vj are tried again.

Figure 4. Thrashing during backtrack

The Waltz filtering algorithm ensures arc consistency in a constraint network. While it falls short of arriving at a complete solution, it is useful for the junction-labeling problem described above in that it leaves only a few ambiguous junctions which can easily be labeled by other means.

A related notion is path consistency, defined as follows [Montanari 1974; Mackworth 1977]:

A path of length m through nodes $(v_{i_0}, v_{i_1}, \ldots, v_{i_m})$ is *path consistent* iff for any value $x \in D_{i_0}$ and $y \in D_{i_m}$ such that $P_{i_0}(x)$ and $P_{i_m}(y)$ and $Q_{i_0 i_m}(x,y)$ there is a sequence of values $z_1 \in D_{i_1}, \ldots, z_{m-1} \in D_{i_{m-1}}$ such that $Q_{i_0 i_1}(x,z_1)$ and $Q_{i_1 i_2}(z_1,z_2)$ and ... and $Q_{i_{m-1} i_m}(z_{m-1},y)$.

A constraint graph is said to be node or arc consistent iff each node or arc, respectively, is consistent. A constraint graph is path consistent iff any pair allowed by any direction relation Q_{ij} is also allowed by all paths from v_i to v_j. Montanari [1974] has shown that if every path of length 2 of a complete graph is path consistent, the graph is path consistent. Since all constraint graphs are implicitly complete graphs (except that arcs are not shown where the universal constraint applies), it suffices to show path consistency for paths of length 2.

We can illustrate node, arc, and path consistency with another blocks puzzle (Figure 5). The puzzle is written as a constraint satisfaction problem below:

$V = \{v_1, v_2, v_3, v_4, v_5, v_6\}$

$D_1, D_2, D_3, D_4, = \{A,B,C,D,E,F\}$

$D_5, D_6 = \{G,H,I\}$

$P_1 = \{(A), (C), (F)\}$

$P_2, P_3 = \{(A), (B), (C), (D), (E), (F)\}$

$P_4 = \{(C), (E), (F)\}$

$Q_{12}, Q_{23}, Q_{34} = \{ (A,B), (B,D), (B,E), (C,A), (C,C), (C,F), (D,B), (E,A), (E,C), (E,F), (F,A), (F,C), (F,F)\}$

$Q_{25}, Q_{46} = \{(A,I), (B,H), (C,I), (D,H), (E,H), (F,G)\}$

$Q_{15}, Q_{36} = \{(A,I), (B,G), (B,H), (C,I), (D,G), (D,H), (E,G), (E,H)\}$

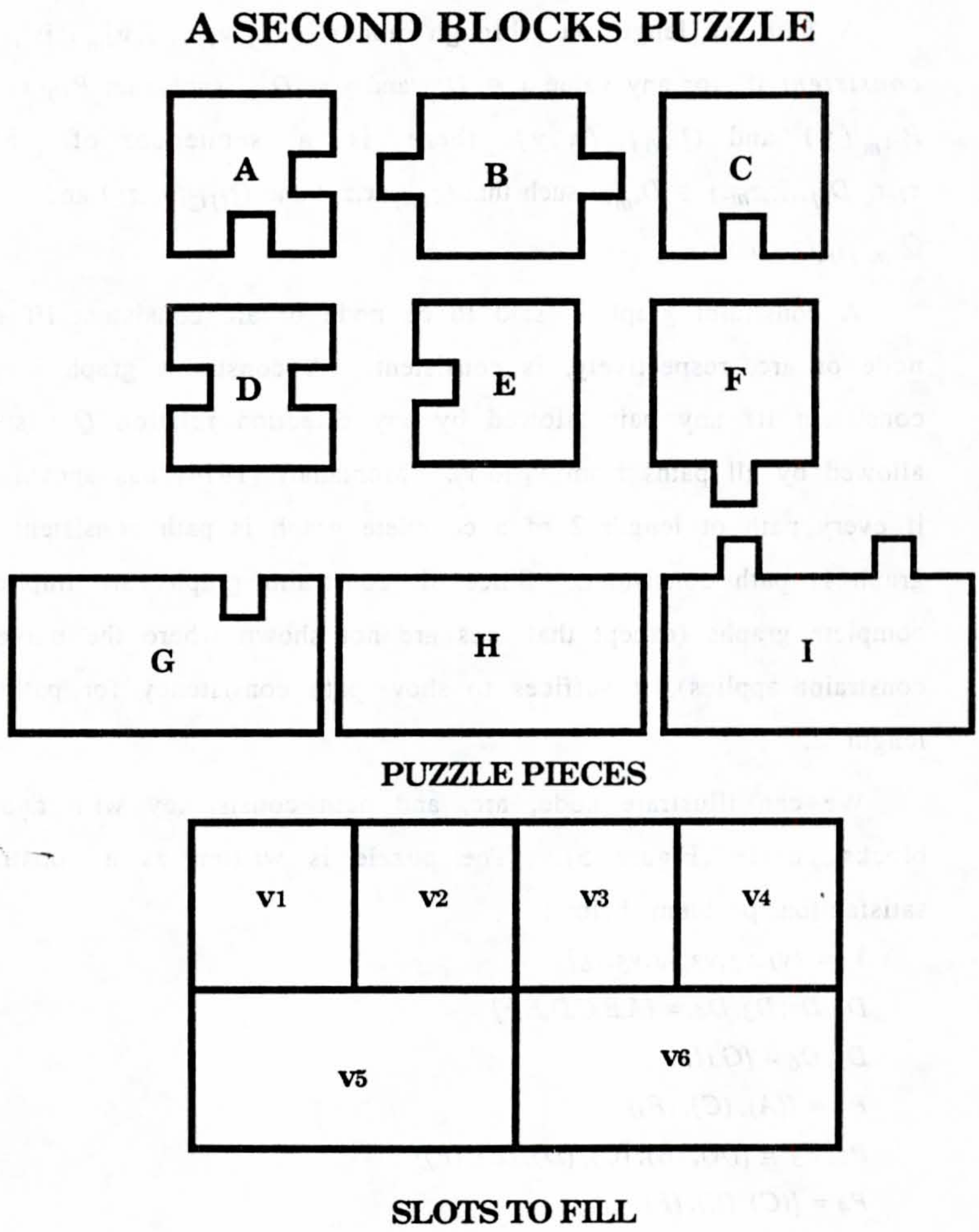


Figure 5. Node, arc and path consistency

To achieve node consistency, we delete *B*, *D*, and *E* from the domain of v_1, and we delete *A*, *B*, and *D* from the domain of v_4.

To achieve arc consistency, we first note that if v_1 is *F*, there is nothing we can put into v_5. Similarly, v_3 cannot be *F*; otherwise we could not label v_6. Thus we delete *F* from the domains of v_1 and v_3.

To achieve path consistency, we note that if $v_1 = A$ and $v_2 = B$, then there is no value for v_5 that is compatible with both v_1 and v_2. Thus we delete *(A,B)* from Q_{12}. Other pairs are deleted in a similar manner. After editing for node, arc, and path consistency, we are left with the following relations:

$Q_{12} = \{(C,A), (C,C)\}$

$Q_{23} = \{(A,B), (B,D), (B,E), (C,A), (C,C), (D,B), (E,A), (E,C), (F,A), (F,C)\}$

$Q_{34} = \{(B,E), (C,C), (E,F)\}$

$Q_{15} = \{(C,I)\}$

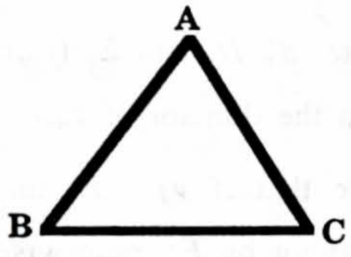
$Q_{25} = \{(A,I), (B,H), (C,I), (D,H), (E,H), (F,G)\}$

$Q_{36} = \{(B,H), (C,I), (E,G)\}$

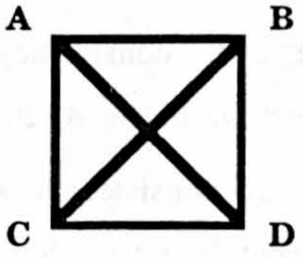
$Q_{46} = \{(C,I) (E,H), (F,G)\}$

2.5.2. k-Consistency

Freuder [1978] noted that arc and path consistency algorithms do not fully synthesize the global constraint. Often additional search is required to find a solution. This can be seen in the example above, where there are two solutions: *(C,A,B,E,I,H)* and *(C,C,C,C,I,I)* for variables 1 through 6 respectively. Furthermore, it is possible for a constraint network to be arc and/or path consistent and still be unsatisfiable, as illustrated in Figure 6.



Color each vertex of the triangle from the domain {RED, GREEN} such that no two adjacent vertices are the same color. (The constraint network is arc consistent but unsatisfiable.)



Color each vertex of the square from the domain {RED, GREEN, YELLOW} such that no two adjacent vertices are the same color. (The constraint network is path consistent but unsatisfiable.)

Figure 6. Consistency and satisfiability

Freuder offers an algorithm which synthesizes the global solution by moving from unary to binary to tertiary and so on to n-ary constraint satisfaction, where n is the number of variables. Freuder's synthesis algorithm uses a process of constraint propagation. For $i = 1$ to n, the algorithm synthesizes i-ary relations, finding consistent i-tuples for these. After k steps of the algorithm, k-consistency is ensured. (Node, arc, and path consistency are 1-, 2-, and 3-consistency, respectively.) If $k = n$, a solution has been found. If $k < n$, a search can be performed to find the remaining values with the assurance that backtrack will not be necessary in the first k levels.

Dechter and Pearl [1988] generalize the notion of k-consistency in an *adaptive consistency* algorithm, in which the level of consistency varies from one node to another, depending upon the order in which variables are instantiated.

2.5.3. A Minimal Constraint Network

Freuder's algorithm and the algorithms for arc and path consistency are based upon the idea of successively eliminating misleading information from the original constraint problem. That is, there may be tuples in the constraint specification which can never contribute to a solution. Such tuples can be eliminated through the logical propagation of information from one constraint to another.

More formally, in problems restricted to binary constraints, the *composition* of two constraints $Q_{ij} \cdot Q_{jk}$ (through Boolean matrix multiplication) *induces* a new constraint Q_{ik}. For the induced constraint Q_{ik}, a tuple (x,z) is in Q_{ik} iff there exists at least one value $y \in D_j$ such that $(x,y) \in Q_{ij}$ and $(y,z) \in Q_{jk}$.

A constraint Q_{ij} is *tighter* than Q'_{ij}, denoted $Q_{ij} \subseteq Q'_{ij}$, iff $(x,y) \in Q_{ij}$ implies $(x,y) \in Q'_{ij}$. A constraint network Q is tighter than a network Q' (denoted $Q \subseteq Q'$) iff, for each pair of corresponding constraints Q_{ij} and Q'_{ij} in the respective networks, $Q_{ij} \subseteq Q'_{ij}$ [Dechter and Pearl 1988].

By inducing new constraints from existing ones, a network can be made tighter and tighter. The tightest network of binary constraints equivalent to a given network is the *minimal network* [Montanari 1974]. Two constraint networks on n variables are said to be *equivalent* if they have the same set of n-tuples for their solutions.

In the minimal network, every tuple in every relation must contribute to some consistent labeling of all n variables. To *minimize* a network, we make the global constraint as local as possible by deleting any tuples which do not participate in some globally consistent labeling.

2.5.4. Time Complexity of Consistency Algorithms

Finding a minimal network clearly must be NP-complete since once a network is minimized, the backtrack-free search requires only $O(ae)$ time (where e is the number of constraints and a is the number of labels) [Montanari 1974].

The arc and path consistency algorithms are polynomial. Mackworth and Freuder [1985] show arc and path consistency algorithms which are $O(ea^3)$ and $O(n^3a^5)$ respectively. Mohr and Henderson [1986] offer faster algorithms of $O(ea^2)$ and $O(n^3a^3)$ respectively (where n is the number of variables). Because each node in the constraint graph can be processed independently, arc and path consistency algorithms also lend themselves to parallel processing [Rosenfeld, Hummel, and Zucker 1976].

2.5.5. Parallel Algorithms for Node, Arc Consistency

From their conception, the node and arc consistency algorithms were described (if not implemented) as parallel algorithms. In the case of node consistency, it is clear that each variable can check its own domain, all variables working in parallel. In the case of arc consistency, the parallelism arises at more than one level. At the largest level of granularity, each variable in parallel with the others can be checking its entire domain. At the next level down, all values in the domain of a given variable can be checked in parallel. Finally, a given value can be checked against all other variables in parallel.

Since the deletion of a value from the domain of a variable could precipitate another deletion of a value from the domain of another variable, the parallel checking must be iterated until no further changes are made.

This inherent parallelism has been realized in a hardware implementation by Swain and Cooper [1987]. Their arc consistency chip consists of two arrays of JK flip-flops and their logical connections. The first array represents the unary constraints P_i and is called the node array. The second array represents the binary constraints Q_{ij} and is called the arc array. The hardware can be used to compute any arc consistency problem where the sizes of P_i and Q_{ij} are bounded by n and a, respectively (Figure 7).

The arc array consists of $a^2n(n-1)$ flip-flops, called $v(i,j,x,y)$ and initialized to $Q_{ij}(x,y)$. That is, if variable i can be labeled x while variable j is labeled y, then flip-flop $v(i,j,x,y)$ is a **1**. (Also, if v_i and v_j do not share a constraint, $v(i,j,x,y)$ is **1**.) The arc array does not change values during the computation.

The node array consists of $a*n$ flip-flops, called $u(i,x)$ and initialized to $P_i(x)$. That is, if x is a permissible label for node i, then the flip-flop $u(i,x)$ is initialized to **1**. The node array is used to keep track of which labels are in the domain of each variable, and at the end of the computation it contains the final answer, i.e., the domains of the variables pruned for arc consistency.

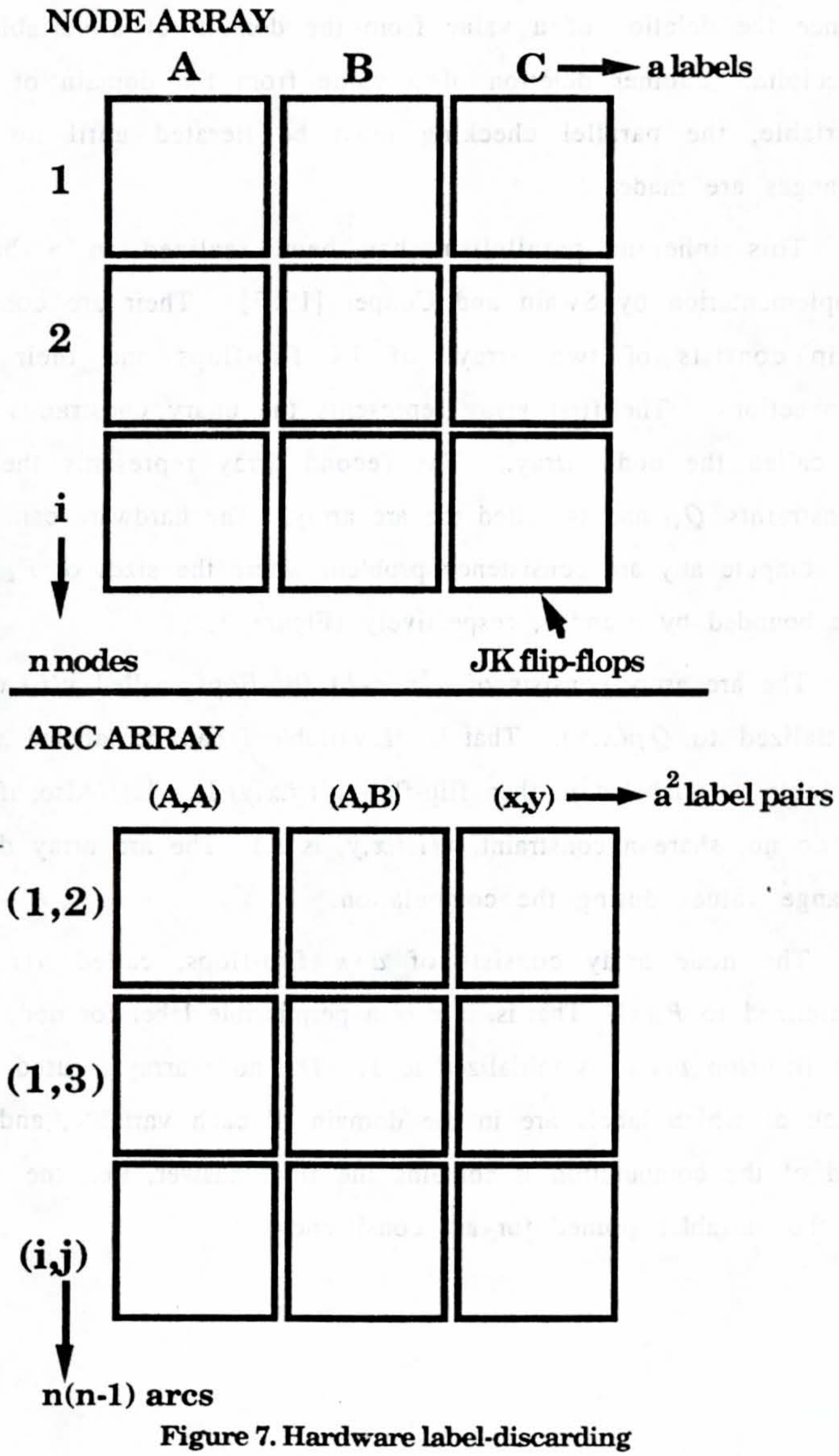


Figure 7. Hardware label-discarding

Additional combinational circuitry is used to implement the label discarding rule. We want the flip-flop representing the label x at node i to be reset to **0** if there exists some other node which has no permissible label while node i is labeled x. The circuitry to accomplish this ties the J (set) input of each JK flip-flop to 0 and uses the K input to reset the flip-flop's value to **0** based on

$$reset(u(i,x)) = \neg \bigwedge_{j=1;j\neq i}^{n} \bigvee_{y=1}^{a} (u(j,y) \wedge v(i,j,x,y)).$$

Figure 8 shows a partial circuit diagram for this equation. It depicts the reset circuitry for flip-flop $u(1,A)$. To determine if A should remain in 1's domain, we must ensure that if 1 is labeled A, there is a label for every other variable which shares a constraint with 1. We are considering only the constraints which variable 1 shares with variables 2 and 3, as indicated in the portion of the arc array shown in Figure 8. Since 1 is mutually constrained with 2 and 3, we must check that variables 2 and 3 have permissible labels. Doing this first for variable 2, we have three $u(j,y) \wedge v(i,j,x,y)$ gates, one for each of the potential labels of variable 2. These in turn are *or-ed* together, since we need only one label for variable 2 which is consistent with the label of A for 1. We check variable 3 similarly, and we *and* them together.

The time complexity of the hardware implementation is $O(an)$ if propagation through the *and* and *or* gates is considered instantaneous. If a logarithmic time cost is assigned to the large fan-in *and* and *or* gates, the complexity is $O(a\log(a)n\log(n))$.

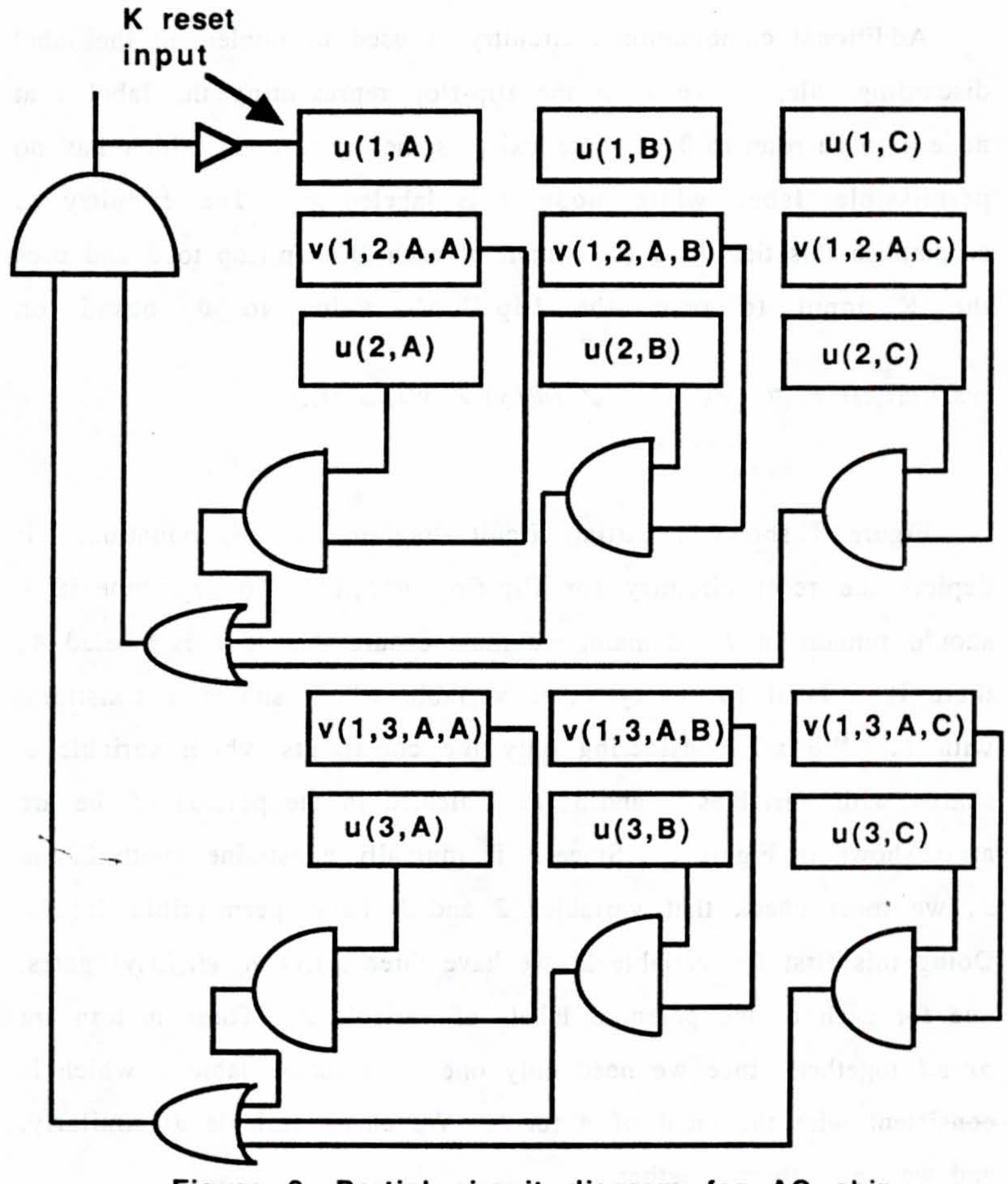


Figure 8. Partial circuit diagram for AC chip

Despite the apparent parallel nature of the consistency algorithms, the speedup achievable by their parallelization appears to have limitations. Kasif [1986] has shown that arc consistency is log-space complete for *P*. This suggests that it is not likely that a poly-log time algorithm will be found. Thus, the *O(an)* for the hardware implementation is likely to be close to the minimum achievable with polynomially-many processors.

Cooper [1988] also describes a connectionist implementation for a subproblem of binary constraint satisfaction, namely, labeled graph matching. Like the filtering algorithm for scene labeling, this hardware implementation does not guarantee a complete matching of the graphs, but is does quickly reject a large number of candidates.

2.5.6. Learning While Searching

Thus far, we have discussed the consistency algorithms as pre-processing procedures. However, they can also be incorporated as procedures within the backtrack-search such that the local consistency of values is checked when an assignment is attempted. When an inconsistency is detected, the reason for it can be determined, and this information can be recorded in some manner. Dechter suggests identifying the conflict set which led to a dead-end in the search and recording it as a new constraint, referring to this technique as *learning while searching* [Dechter 1986; Dechter 1988].

Learning within a constraint network can also be considered from another point of view. From this perspective, a constraint problem models a world or an object of which we have only incomplete, uncertain knowledge. The constraints represent the current state of our knowledge about the world being modeled, and it is subject to revision. As new facts come to light, we enter them into the constraint specification and observe their implications. This view of a constraint problem leads to a dynamic constraint network which is satisfied incrementally when new knowledge (i.e. "learning") comes in from an external source (i.e. the user).

Within this scheme, Dechter and Dechter [1987] devise an algorithm for computing the support for a label x of a variable v_i. Through a process of support propagation, the algorithm computes the number of solutions in which the labeling $v_i = x$ participates. This value represents the strength of our belief that $v_i = x$. When a new fact is entered into the system, say $v_j = y$, the system enters a stage of contradiction resolution. This algorithm identifies the minimum number of assumptions that must be changed in order to restore consistency. (Some variables in the network are designated as assumption variables which are initially assigned default values. These may be assigned other values to resolve contradictions.) Both the support propagation and the contradiction resolution algorithms lend themselves to parallel implementation.

2.5.7. The $\varphi_{k,p}$ Operator

The notions of arc and path consistency have been extended from binary to N-ary relations in Haralick and Shapiro's work with the consistent labeling problem [1979]. Like Dechter's learning algorithm, their tree-search-reduction procedure is executed during the search rather than as a preprocessing step.

With the $\varphi_{k,p}$ operator, we check in turn each N-tuple of R. For each such N-tuple, we fix a subset of k units to their labels in the tuple, and we check to see if that labeling can be extended to p units for every possible combination of $p - k$ remaining units. In this context, arc consistency for N-ary relations is given by $\varphi_{1,N}$, while path consistency is $\varphi_{2,3}$.

Consider applying the $\varphi_{2,4}$ operator to the tuple $(1,A,3,B)$ in the consistent labeling problem below, with $k = 2$ and $p = 4$:

$U = \{1,2,3,4,5\}$
$L = \{A,B,C\}$
$T = \{(1,2), (1,3), (1,4), (1,5), (2,3), (2,4), (2,5), (3,4), (3,5), (4,5)\}$
$N = 2, M = 5$
$R = \{$ $(1,A,2,A)$, $(1,A,2,B)$, $(1,A,3,A)$, $(1,A,3,B)$, $(1,A,3,C)$, $(1,A,4,A)$, $(1,A,4,B)$, $(1,A,4,C)$, $(1,A,5,A)$, $(2,A,3,A)$, $(2,A,3,B)$, $(2,B,3,C)$, $(2,A,4,A)$, $(2,A,4,B)$, $(2,A,4,C)$, $(2,B,4,C)$, $(2,A,5,A)$, $(2,B,5,A)$, $(3,A,4,A)$, $(3,A,4,B)$, $(3,B,4,A)$, $(3,C,4,A)$, $(3,C,4,C)$, $(3,B,5,A)$, $(3,B,5,B)$, $(3,C,5,A)$, $(4,C,5,A)$, $(4,C,5,B)\}$

This entails fixing 1 to the label A and 3 to the label B, and checking that this labeling can be extended to a consistent labeling for the subsets of units $\{1,2,3,4\}$, $\{1,2,3,5\}$, and $\{1,3,4,5\}$. We find there exists a consistent labeling for the subset $\{1,2,3,4\}$: $\{A,A,B,A\}$. Similarly, $\{A,A,B,A\}$ is a consistent labeling for the subset $\{1,2,3,5\}$. However, there is no consistent labeling possible for the subset $\{1,3,4,5\}$ if 1 and 3 are fixed at A and B respectively. Thus the tuple $(1,A,3,B)$ is deleted from R.

2.5.8. Domain-Pruning Strategies

Arc and path consistency algorithms, the $\varphi_{k,p}$ operator, and Waltz filtering are examples of lookahead operators. Haralick and Elliott [1980] have classified strategies for reducing the tree search of binary constraint problems into full looking ahead, partial looking ahead, forward checking, backchecking, and backmarking, and they give search algorithms which incorporate these strategies.

Looking ahead is a technique for trimming the domains of the units not yet labeled (called *future units*) at each point in the search tree. Each time a label assignment is made, the *current unit* (the unit just labeled) is checked for consistency against all future units. That is, the domains of future units are reduced to include only those labels consistent with the current unit. Then on the basis of the restricted domains, every future unit is checked against every other future unit. If there exists a label x for future unit u_i which does not have a compatible label for some other future unit u_j, then x is deleted from u_i's domain.

Looking ahead is illustrated by the labeling problem in Section 2.5.7. D_i denotes the set of labels (the domain) available for unit i. The units $\{1,2,3,4,5\}$ are to be labeled in numerical order. Unit 1 is assigned the first label in its domain, A. In the current-future consistency checks, we see that if 1 is A, then 2 must be A or B; 3 can be A, B or C; 4 can be A, B, or C; and 5 must be A. The domains are restricted accordingly.

In the future-future checks, we first check 2 against 3, 4, and 5. Unit 2's domain has been restricted to $\{A,B\}$ in the current-future checks. If 2 is A, 3 can be A or B; 4 can be A, B, or C; and 5 can be A. If 2 is B, 3 can be C; 4 can be C, and 5 can be A. Thus there are labels available for all other future units if 2 is labeled A or B, and 2's domain does not change. Checking 3 against 2, 4, and 5, we find that if 3 is A, there is no label available for 5. Thus, 3's domain is restricted to *{B,C}*. In a similar manner, *4*'s domain is restricted to *{C}*. Since labels remain in the domains of all units, the label of *A* for *1* has been verified, and we continue down the search tree.

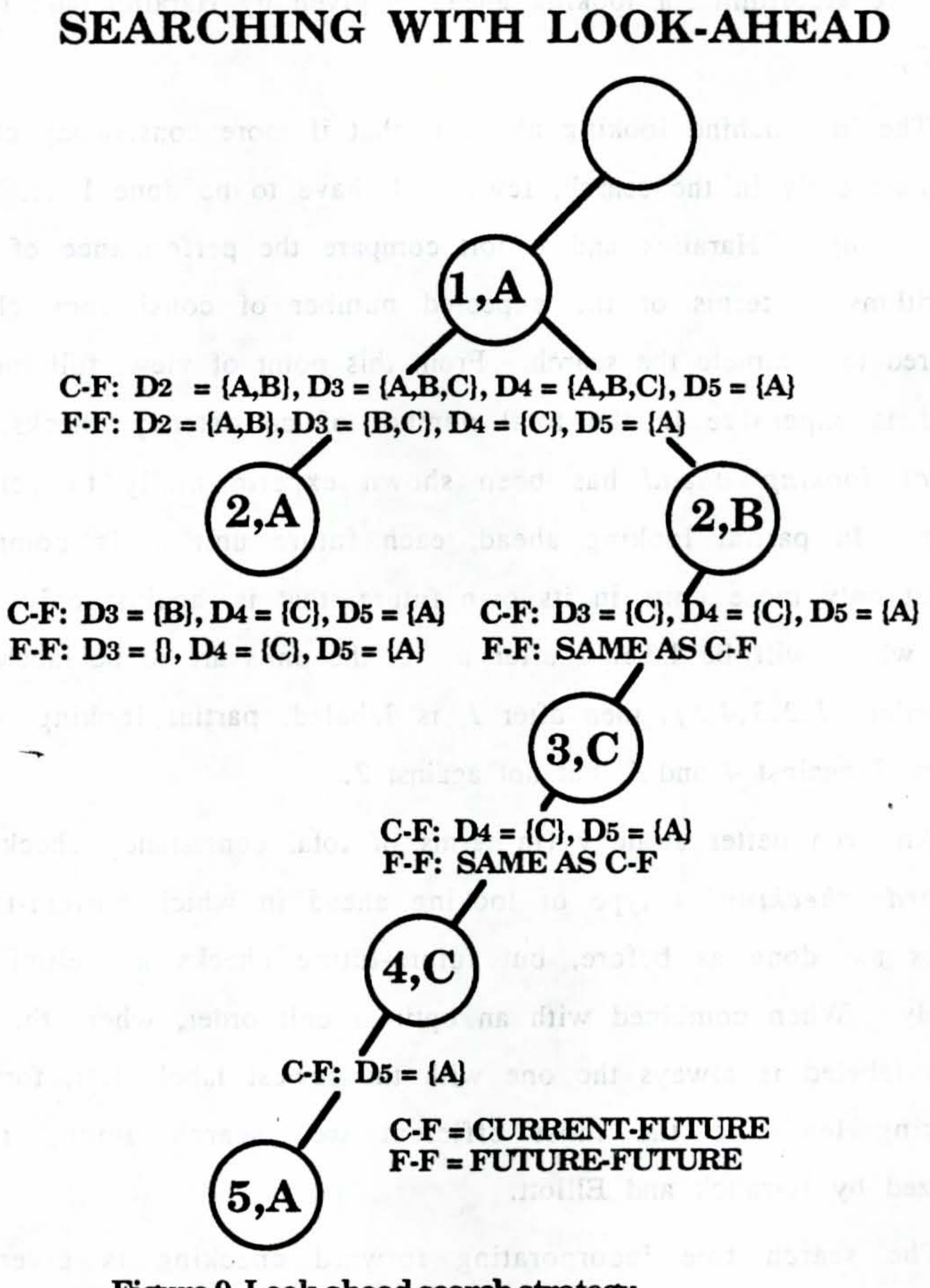

SEARCHING WITH LOOK-AHEAD
1,A
C-F: D2 = {A,B}, D3 = {A,B,C}, D4 = {A,B,C}, D5 = {A}
F-F: D2 = {A,B}, D3 = {B,C}, D4 = {C}, D5 = {A}
2,A
2,B
C-F: D3 = {B}, D4 = {C}, D5 = {A}
F-F: D3 = {}, D4 = {C}, D5 = {A}
C-F: D3 = {C}, D4 = {C}, D5 = {A}
F-F: SAME AS C-F
3,C
C-F: D4 = {C}, D5 = {A}
F-F: SAME AS C-F
4,C
C-F: D5 = {A}
C-F = CURRENT-FUTURE
F-F = FUTURE-FUTURE
5,A

Figure 9. Look-ahead search strategy

The entire search tree, which incorporates looking ahead, is given in Figure 9. The arcs are labeled with the restricted domains resulting from the current-future and future-future look-aheads. A recursive algorithm for looking ahead is given by Haralick and Elliott [1980].

The idea behind looking ahead is that if more consistency checks are done early in the search, fewer will have to be done later, for a total savings. Haralick and Elliott compare the performance of their algorithms in terms of the expected number of consistency checks required to complete the search. From this point of view, full looking ahead is expensive in the total number of consistency checks, and *partial looking ahead* has been shown experimentally to perform better. In partial looking ahead, each future unit *u* is compared against only those units in its own future, that is, against only those units which will be labeled after *u*. If the units are to be labeled in the order *{1,2,3,4,5}*, then after *1* is labeled, partial looking ahead checks *3* against *4* and *5*, but not against *2*.

An even better strategy (in terms of total consistency checks) is *forward checking*, a type of looking ahead in which current-future checks are done as before, but future-future checks are eliminated entirely. When combined with an optimal unit order, where the unit to be labeled is always the one with the fewest labels left, forward checking leads to the most efficient tree search among those analyzed by Haralick and Elliott.

The search tree incorporating forward checking is given in Figure 10. Note that the search tree with looking ahead is smaller than the one with forward checking, but at the expense of more consistency checks.

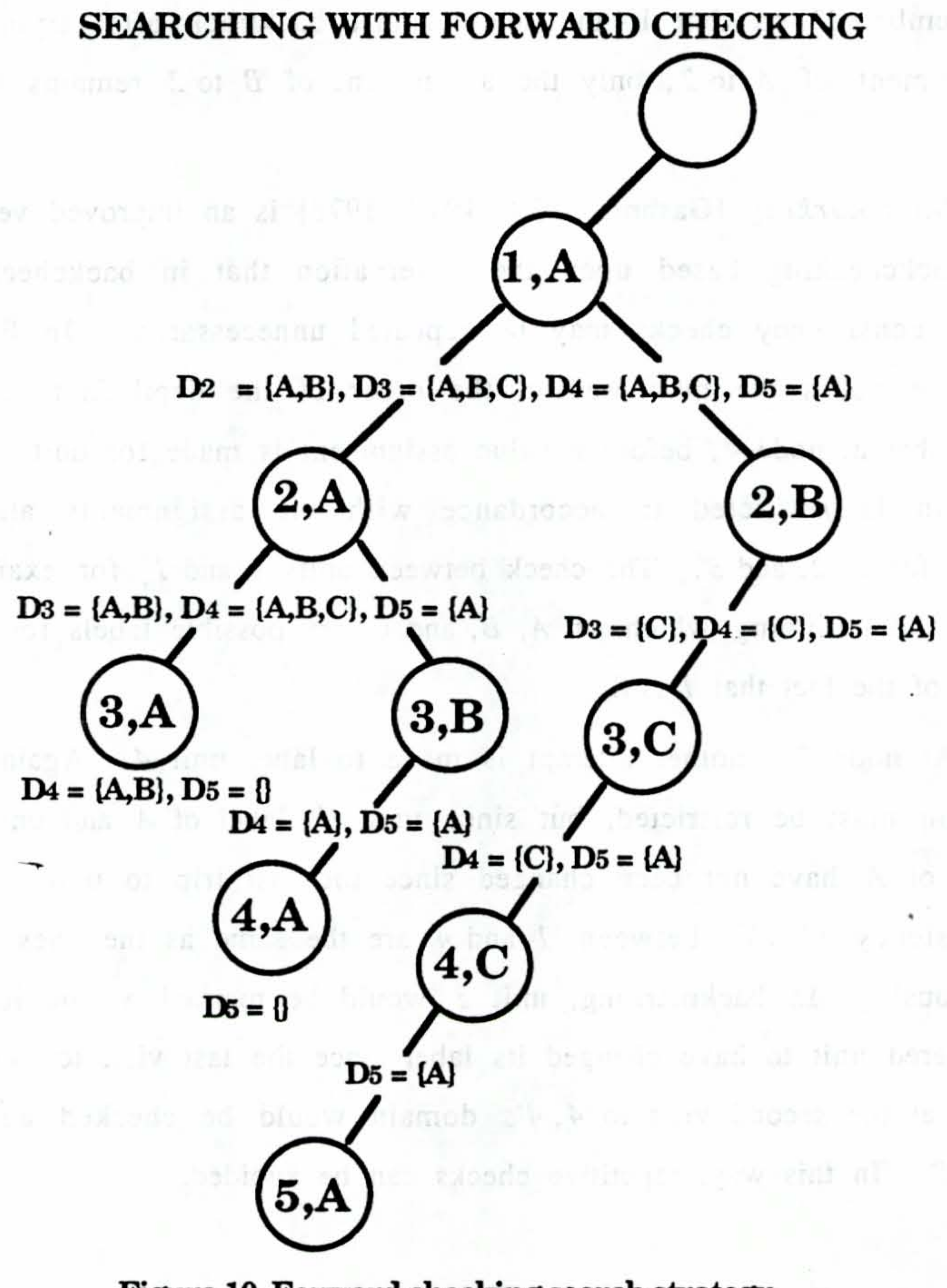

SEARCHING WITH FORWARD CHECKING
1,A
D2 = {A,B}, D3 = {A,B,C}, D4 = {A,B,C}, D5 = {A}
2,A
2,B
D3 = {A,B}, D4 = {A,B,C}, D5 = {A}
D3 = {C}, D4 = {C}, D5 = {A}
3,A
3,B
3,C
D4 = {A,B}, D5 = {}
D4 = {A}, D5 = {A}
D4 = {C}, D5 = {A}
4,A
4,C
D5 = {}
D5 = {A}
5,A

Figure 10. Forward checking search strategy

Backchecking differs from looking ahead in that domains are restricted only with respect to label assignments already made. For example, in Figure 11 we see that once *1* has been given the label *A*, *2*'s domain can be restricted to *{A,B}*. This information is "remembered" so that if the search must backtrack after trying an assignment of *A* to *2*, only the assignment of *B* to *2* remains to be tried.

Backmarking [Gashnig 1974; 1977; 1978] is an improved version of backchecking based upon the observation that in backchecking, some consistency checks may be repeated unnecessarily. In Figure 11, the nodes are numbered in the order of the depth-first search. Note that at node *4*, before a value assignment is made for unit *4*, its domain is restricted in accordance with the assignments already made for *1*, *2*, and *3*. The check between units *4* and *1*, for example, entails determining which of *A*, *B*, and *C* are possible labels for *4* in view of the fact that *1* is *A*.

At node *7*, another attempt is made to label unit *4*. Again, *4*'s domain must be restricted, but since unit *1*'s label of *A* and unit *2*'s label of *A* have not been changed since the last trip to unit *4*, the consistency checks between *1* and *4* are the same as the ones done previously. In backmarking, unit *3* would be marked as the lowest numbered unit to have changed its label since the last visit to unit *4*. Then at the second visit to *4*, *4*'s domain would be checked against only *3*. In this way, repetitive checks can be avoided.

SEARCHING WITH BACKCHECKING

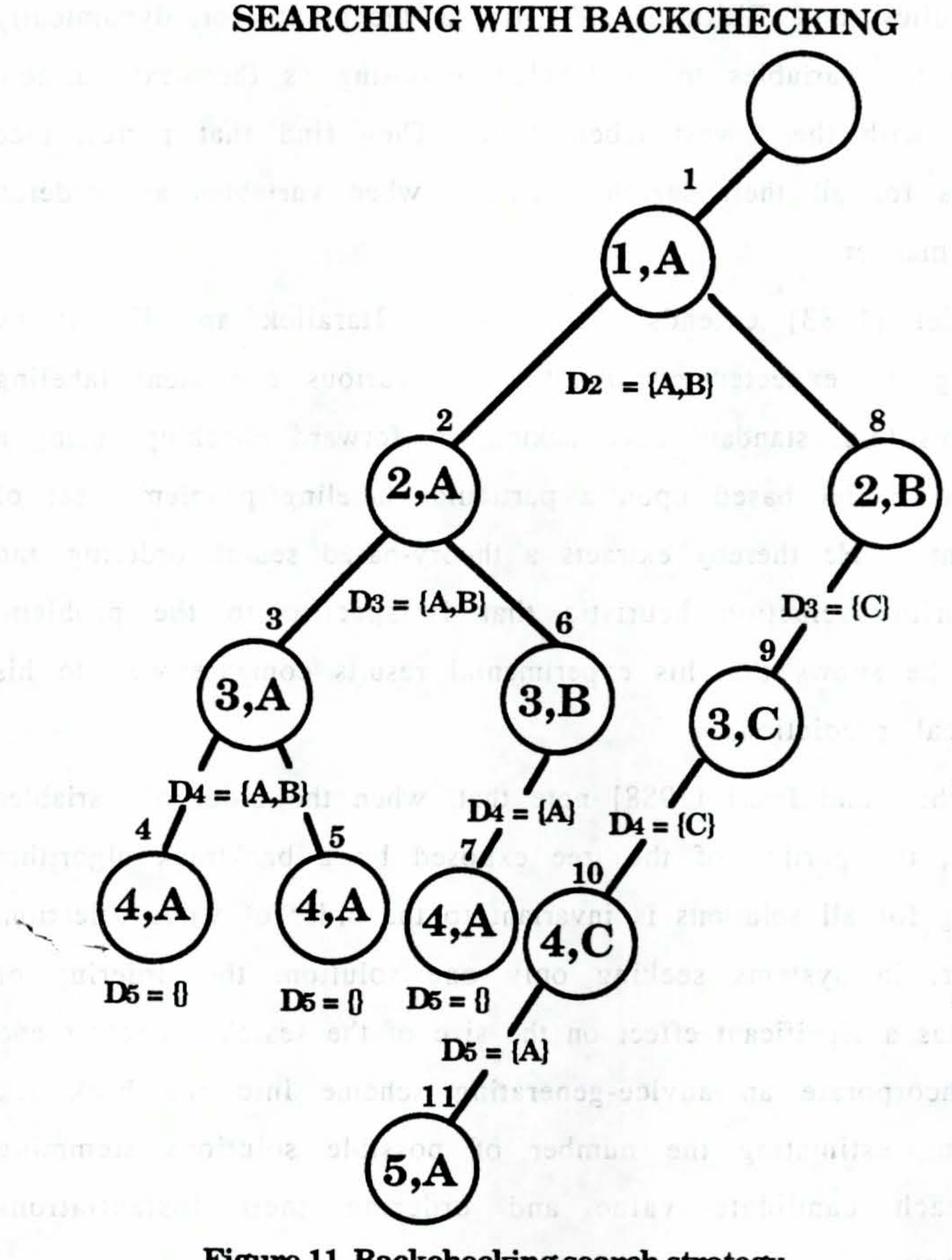


Figure 11. Backchecking search strategy

2.5.9. Ordering Variable and Value Assignments

Both the order in which variables will be considered and the order in which values will be tried affect the size of the search in constraint satisfaction.

Haralick and Elliott [1980] try a technique for dynamically ordering the variables to be labeled, choosing as the next variable the one with the fewest labels left. They find that performance improves for all their search strategies when variables are ordered in this manner.

Nudel [1983] extends the work of Haralick and Elliott by obtaining the expected complexities for various consistent labeling algorithms (e.g. standard backtracking vs. forward checking) using a statistical model based upon a particular labeling problem's set of constraints. He thereby extracts a theory-based search ordering and an algorithm-selection heuristic that is specific to the problem. Finally, he shows that his experimental results compare well to his theoretical predictions.

Dechter and Pearl [1988] note that, when the order of variables is fixed, the portion of the tree exposed by a backtrack algorithm searching for all solutions is invariant to the order of value selection. However, in systems seeking only one solution, the ordering of values has a significant effect on the size of the search. Dechter and Pearl incorporate an advice-generating scheme into the backtrack algorithm, estimating the number of possible solutions stemming from each candidate value and ordering their instantiations accordingly.

2.6. Discrete Relaxation and Relaxation Labeling

In vision applications, the arc consistency algorithm is referred to as a kind of *discrete relaxation* or *relaxation labeling*. The term *relaxation* is used because of the algorithm's similarity to the iterative processes used in numerical analysis.

Davis and Rosenfeld [1981] describe the relaxation processes used in vision problems as follows:

1. A list of possible labels is selected for each part of an image. For example, we may be labeling a pixel as part of an edge, a corner, an interior point, or an exterior point. At a higher level of abstraction, we may be labeling an image segment as a table, a chair, or a bed. A measure of confidence (a weight) can also be associated with each label. (Weighted labels are sometimes referred to as fuzzy labels.)
2. In a parallel, iterative fashion, the labels for each part are compared with those for related parts, based upon our knowledge of how things can "fit together." Labels are deleted or weights adjusted to reduce inconsistencies.

To accomplish the relaxation, we need to specify the neighboring (i.e. mutually constrained) variables, and how the labels of one variable will change with respect to its neighbors' labels. The simplest relaxation mechanism is the label discarding rule implemented in the arc consistency algorithm.

Hummel and Zucker [1983] extend the relaxation procedure to a procedure for *continuous relaxation labeling*, where both label assignments and constraints are given weights. The weight with which label x is assigned to variable v_i is denoted $P_i(x)$ (where $0 \leq P_i(x) \leq 1$), while the relative support for label x at variable v_i that arises from label x' at variable v_j is denoted by the real-valued compatibility function $Q_{ij}(x,x')$. On this foundation, Hummel and Zucker attempt to build a more general theory of consistency using variational calculus and standard optimization techniques.

Another variation of the original Waltz filtering algorithm works upon variable domains expressed as real-number intervals [Davis 1987]. For example, the domains of v_1, v_2, and v_3 may be expressed as:

$v_1 \in [1,10]$, $v_2 \in [3,8]$, $v_3 \in [2,7]$,

and the constraints may be expressed as

$v_1 + v_2 = v_3$, $v_2 \leq v_1$.

Then by a process of label refinement, the label set of each node is restricted in accordance with the domains of its neighbors. After applying the numerical equivalent of the Waltz filtering algorithm, we are left with $v_1 \in [3,4]$, $v_2 \in [3,4]$, and $v_3 \in [6,7]$.

While this procedure goes by the name of *interval labeling* and borrows a technique from finite-domain constraint satisfaction, it defines constraints over real-number domains and for this reason perhaps is more properly classed with the numerical constraint satisfaction methods to be discussed in Section 3.

3. CONTINUOUS-DOMAIN CONSTRAINT SATISFACTION

3.1. Numerical Constraints: Modeling, Problem-Solving

A constraint problem as a labeling problem tells us which discrete parts in a domain of finite elements "fit together." Continuous-domain (i.e. numerical) constraints, on the other hand, are expressed as equalities or inequalities defined over a real-number domain.

In modeling and simulation, numerical constraints might describe the physical subsystems of an object, involving properties such as mass, density, pitch, color, current, voltage, position, velocity, and the like. In this way, continuous-domain constraints are a powerful modeling tool.

Numerical constraints lend themselves to the definition of geometric objects since they readily describe the relative positions of an object's parts. Section 3.3.2 reviews a number of these applications. Numerical constraints are also applicable to the description of an object as a physical system, e.g. an electrical circuit, as discussed in Section 3.3.3. Finally, constraints can be used to model systems in a physically realistic manner, behaving in accordance with forces, torques, energies, and the laws of Newtonian physics. The application of constraints to physically-based modeling is discussed in Section 3.3.4.

We begin by reviewing the basic methods for numerical constraint satisfaction.

3.2. Techniques for Numerical Constraint Satisfaction

3.2.1. Propagation of Known States

An equality or inequality defines a constraining relationship among the variables it involves. For example, the equation

$\pi * d = c$

expresses a constraint between the diameter and circumference of a circle. Values which abide by this relation are implicit in the semantics of the equation. Given a value for c, we can derive d, and vice versa.

In his review of constraint-based programming, Leler [1988] suggests a graphical representation of numerical constraints to illustrate the operation of propagation of known states (also referred to as *local propagation*). (In this section, we will be referring to the type of graph depicted below when we speak of a constraint graph, as opposed to the constraint networks described in Section 2.5.)

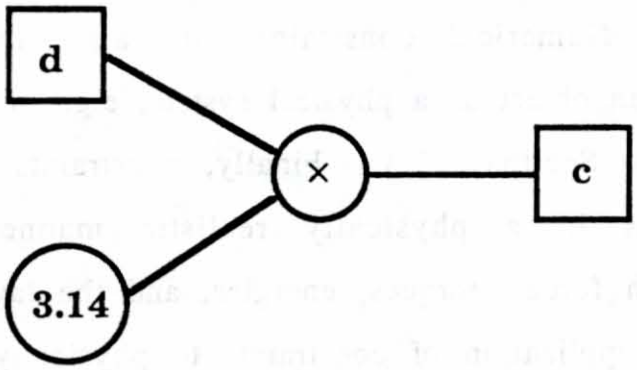


Figure 12. Constraint graph for circumference

Squares in Figure 12 represent variables, and circles represent either operators or constants. The arguments to the operator appear on the left, while the result is placed on the right (with no = sign required). *Propagation of known states* is a procedure by which known values are used to compute new values in a constraint equation. It should be emphasized that values can propagate in either direction. It is up to the constraint satisfaction system to recognize when a sufficient number of variables have been bound to values and to understand the semantics of the operators in either direction. Thus, if c is known in the constraint above, a constraint satisfaction system based upon local propagation should be able to compute *d* by dividing *c* by *3.14*.

Propagation of known states is the simplest of the constraint satisfaction methods. Another advantage to this method is that the path by which a particular answer was produced can be recorded. Thus the user can request explanations of an answer, and can even roll back the solution to an earlier point, continuing from there without requiring the system to re-evaluate the entire problem. However, local propagation is of limited use since it considers only information local to a node, and thus it cannot be applied to constraints involving cycles.

Consider the following constraints:

$A + B = C$

$B + C = D$

Because of the interdependency of *B* and *C*, these constraints form a cycle, as represented in Figure 13.

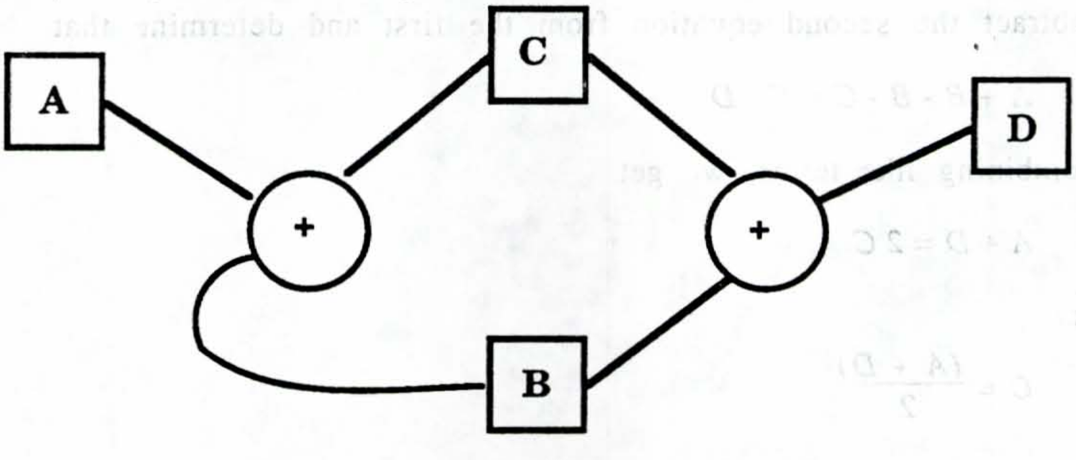


Figure 13. Constraint graph with a cycle

Local propagation cannot find a solution to this set of constraints. To see this, consider starting with $A = 5$, $B = -1$, $C = 4$, $D = 3$. This satisfies the constraints since $A + B = C = 4$ and $B + C = D = 3$. Now, what happens if we change A to 9 and attempt to keep D unchanged? Using propagation, we could compute $C = A + B = 9 - 1 = 8$. We could then compute $B = D - C = 3 - 8 = -5$, and come to a conflict with B having values -1 and -5.

If we accept the latest value of B, -5, and propagate this, we will get $C = 4$ and then $B = -1$. B's value is flipping back and forth in a simple cycle. Relaxation (Section 3.2.3) can be used to try a compromise, $B = -3$, the midpoint of -1 and -5. Alternatively, algebraic manipulation can be used to produce solutions even in the presence of cycles.

3.2.2. Algebraic Manipulation and Term Rewriting

Looking back at Figure 13, we can see that this constraint might be resolved by algebraic simplification. In particular, we could subtract the second equation from the first and determine that

$$A + B - B - C = C - D .$$

Combining like terms, we get

$$A + D = 2\,C$$

or

$$C = \frac{(A + D)}{2}.$$

In other words, C is just the average of A and D.

One way to make algebraic transformations of subparts of a constraint expression is by means of a *term rewriting systems* (Section 3.3.1.) In a term rewriting system, expressions can be rewritten according to a set of rewrite rules. Each rewrite rule consists of a head (the left-hand side) and a body (the right-hand side), e.g.

$$\underset{\substack{\uparrow \\ \textit{head}}}{X - X} \Rightarrow \underset{\substack{\uparrow \\ \textit{body.}}}{0}$$

We let uppercase characters in the rewrite rules denote variables, which can be matched to any subexpression. When the head of a rewrite rule matches a subexpression of an expression E, that subexpression can be rewritten in accordance with the body of the rule. For example, given the rewrite rule

$X - X \Rightarrow 0$

we can match the head of the rule to the subexpression $B - B$ in $A + B - B - C = C - D$, and rewrite the entire expression to $A + 0 - C = C - D$. Then, given the rewrite rule

$X + 0 \Rightarrow X$

we can rewrite the equation to $A - C = C - D$.

Term rewriting can be used to transform a simple cycle which could not otherwise be handled by local propagation. However, cycles formed by simultaneous equations cannot be broken by standard term rewriting techniques. For these we need numerical relaxation techniques or more sophisticated equation solvers.

3.2.3. Relaxation

A number of iterative numerical methods have been devised for solving systems of linear equations [Maron 1982]. These methods are applicable to cyclical constraint sets which constitute a system of linear equations.

Consider a system of linear equations, which take the form:

$$\begin{array}{l}
a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = b_1 \\
a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n = b_2 \\
\vdots \\
a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n = b_n.
\end{array}$$

One method of solving for the unknowns is to make initial guesses, estimate the errors that would result from these values, derive new guesses based on the errors, and iterate until the method converges on a solution. The Gauss-Seidel procedure uses this strategy. It begins with initial values for x_j, $1 \leq j \leq n$, and repeatedly computes $x_j^{(new)}$ from x_j using the i-th equation, according to:

$$x_j^{(new)} = x_j + \frac{1}{a_{ij}} \{b_i - \sum_{k=1}^{n} a_{ik}x_k\}.$$

(To improve chances of convergence, it is best to solve as many equations as possible for the variable having the greatest-magnitude coefficient.)

This method can be viewed as adding some increment dx_j to x_j in order to get $x_j^{(new)}$:

$$x_j^{(new)} = x_j + dx_j$$

where $dx_j = \frac{1}{a_{ij}} \{b_i - \sum_{k=1}^{n} a_{ik}x_k\}$. The increment dx_j indicates how well x_j satisfies the *i-th* equation, and the change in dx_j indicates how well the algorithm is converging. Convergence is not guaranteed for the Gauss-Seidel method.

A technique called *relaxation* is sometimes used to improve the likelihood or rate of convergence. Relaxation works by solving the *i-th* equation for x_i and multiplying the increment by some λ_i:

$$x_i^{(new)} = x_i + \lambda_i dx_i.$$

When $0 < \lambda_i < 1$, *underrelaxation* is being used, and when $1 < \lambda_i < 2$, *overrelaxation* is being used. A value of $\lambda_i = 1$ provides no relaxation and commonly leads to non-convergence.

To see how relaxation works, we return to solving the constraints of Figure 13. In keeping with our standardized forms for systems of linear equations, we can rewrite the constraints as

$C - B = A$, and [1]

$B + C = D$, [2]

where A and D are fixed as the constants 9 and 3. Our initial guesses for B and C are

$$B^{(1)} = -1 \quad \textit{and} \quad C^{(1)} = 4.$$

Using [1] as the basis for computing B and [2] as the basis for C, we get new approximations by the formulas

$B^{(new)} = B - \lambda_B(9-(C-B))$ **and**

$C^{(new)} = C + \lambda_C(3-(B+C))$.

If we set $\lambda_B = \lambda_C = 1$ (no relaxation), we get successive approximations as follows:

$B^{(1)} = -1$ $\qquad$ $C^{(1)} = 4$

$B^{(2)} = -1 - (9-5) = -5$ $\qquad$ $C^{(2)} = 4 + (3+1) = 8$

$B^{(3)} = -5 - (9-13) = -1$ $\qquad$ $C^{(3)} = 8 + (3-7) = 4$

and we are in an infinite loop. Setting $\lambda_B = \lambda_C = \frac{1}{2}$ works much better:

$B^{(1)} = -1$ $\qquad$ $C^{(1)} = 4$

$B^{(2)} = -1 - \frac{1}{2}(9-5) = -3$ $\qquad$ $C^{(2)} = 4 + \frac{1}{2}(3-1) = 5$

$B^{(3)} = -3 - \frac{1}{2}(9-8) = -3\frac{1}{2}$ $\qquad$ $C^{(3)} = 5 + \frac{1}{2}(3 - 1\frac{1}{2}) = 5\frac{3}{4}$

$B^{(4)} = -3\frac{1}{2} - \frac{1}{2}(9 - 9\frac{1}{4}) = -3\frac{3}{8}$ $\qquad$ $C^{(4)} = 5\frac{3}{4} + \frac{1}{2}(3 - 2\frac{3}{8}) = 6\frac{1}{16}$

$B^{(5)} = -3\frac{3}{8} - \frac{1}{2}(9 - 9\frac{7}{16}) = -3\frac{5}{32}$ $\qquad$ $C^{(5)} = 6\frac{1}{16} + \frac{1}{2}(3 - 2\frac{29}{32}) = 6\frac{7}{64}$

$B^{(6)} = -3\frac{5}{32} - \frac{1}{2}(9 - 9\frac{17}{64}) = -3\frac{3}{128}$ $\qquad$ $C^{(6)} = 6\frac{1}{64} + \frac{1}{2}(3 - 3\frac{11}{128}) = 5\frac{249}{256}$

Yet another approach is to relax just one variable, say B. Here we will set $\lambda_B = \frac{1}{2}$ and $\lambda_C = 1$ (no relaxation).

$B^{(1)} = -1$ $\qquad$ $C^{(1)} = 4$

$B^{(2)} = -1 - \frac{1}{2}(9-5) = -3$ $\qquad$ $C^{(2)} = 4 + (3-1) = 6$

$B^{(3)} = -3 - \frac{1}{2}(9-9) = -3$ $\qquad$ $C^{(3)} = 6 + (3-3) = 6.$

For this particular example, convergence was achieved quickly by the last choice of relaxation constants. In general, selecting good values is a non-trivial problem. In fact, even the choice of which variables to satisfy is not always obvious.

The next section further addresses the issue of which variables might be advantageously relaxed.

3.2.4. Propagation of Degrees of Freedom

The relaxation approach just discussed can be inefficient if we choose to relax variables which do not participate in cycles. A better approach is to prune the constraint graph until the only nodes left are those tied up in cycles. This technique, called *propagation of degrees of freedom*, counts the number of constraints applied to each node. A node with no freedom (a constant) can be ignored for now, since its value is fixed. One with just one constraint also represents an easy case. Such a node cannot be in a cycle, and hence its value is determinable by simple propagation of known values along its one constraint.

To isolate the cycles, we remove constant nodes first. We then remove those controlled by just one constraint. Removal of such a node reduces the number of constraints on its neighbors, and so we can continue this process. Finally, the cycles are identified and can be resolved by relaxation methods.

Looking back at Figure 13, we can see that *A* and *D* are involved in just one constraint each. Removing them leaves us with a two-node cycle on which we must perform some cycle-breaking technique such as relaxation or algebraic manipulation.

A side benefit of propagation of degrees of freedom is that it provides a plan for propagating values after the cycle is broken. This plan is just propagation of values through the nodes just removed, in reverse chronological order (the last removed shall be the first to be evaluated except, of course, for constants which have already been evaluated.) Thus, propagation of degrees of freedom can result in a pre-compiled constraint method for the satisfaction of a set of constraints.

3.2.5. Constrained Optimization, Linear Programming

While constraint-based programming has only recently become a field of research in its own right, the idea of expressing a problem quantitatively in terms of constraints is nothing new. The recognition that many real-world problems can be expressed in terms of constraints led to the development of classical optimization theory and its application in fields as diverse as classical mechanics [Goldstein 1965] and operations research [Shamblin and Stevens 1974].

In the basic *constrained optimization problem*, the goal is to minimize or maximize an objective function

$f(x_1, x_2, \ldots, x_n)$

given a number of constraint functions of the form

$$\begin{array}{ccc} g_1(x_1, x_2, \ldots, x_n) \le b_1 & & \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ g_m(x_1, x_2, \ldots, x_n) \le b_m & & \end{array}$$

In a constrained optimization problem, we want not only values which are consistent with all the constraints, but in some sense the "best" such values. If all the functions in the problem are linear functions, we have what is known as the *linear programming* *problem*, the most widely-used form of mathematical constrained optimization.

As an example, consider the following practical problem: A company produces two products, x_1 and x_2. The profit for product x_1 is $100 per unit, while the profit for x_2 is $150. One unit of product x_1 requires 20 hours of production time, while one unit of x_2 requires 35 hours. The total time available for manufacturing in one year is 1500 hours. Based on expected demands, it has been determined that not more than 42 units of x_1 and not more than 30 units of x_2 should be produced each year. The goal is to maximize the profit. The problem is modeled by the objective function

$$100x_1 + 150x_2$$

and the constraint functions

$$20x_1 + 35x_2 \leq 1500,$$
$$x_1 \leq 42, \text{ and}$$
$$x_2 \leq 30.$$

One well-known method for solving the linear programming problem is the *simplex algorithm*. The simplex algorithm takes advantage of the facts that the objective function is linear, and the intersections of the constraints form a convex polyhedron. This is easy to picture in the case of a two-dimensional function, as shown in Figure 14. (Since it does not make sense for x_1 and x_2 to be negative, we need consider only the non-negative quadrant of two-dimensional Euclidean space.)

It is also known that if an optimum solution exists, at least one of the vertices of the polyhedron will be an optimum solution. In

Figure 14, the point (42, 18.9) represents the optimum value for the objective function within the feasible solution space.

The simplex algorithm begins by converting the inequalities to equalities by means of *slack variables*. In this example we get:

$$20x_1 + 35x_2 + u_1 = 1500$$
$$x_1 + u_2 = 42$$
$$x_2 + u_3 = 30,$$

with slack variables u_1, u_2, and u_3. We now have a system of linear equations, but we have fewer equations than unknowns.

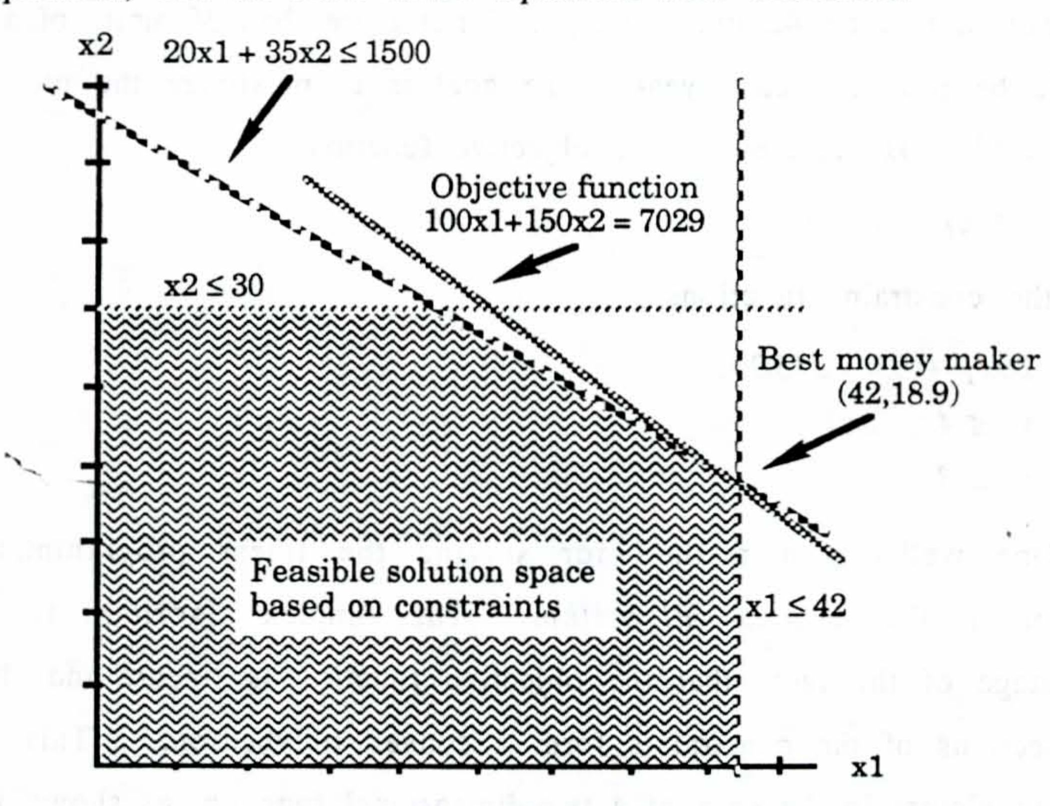


Figure 14. Simplex method checks vertices only

If we test all the vertices of the polyhedron, we will eventually find an optimum solution. In this example, this can be done by setting two of the five variables to 0 (for every possible pair of variables) and solving the resulting system of linear equations. The variables in solution are called *basis variables*, and the variables set to 0 are called *non-basis variables*.

Since all of our inequalities are $\leq$, we can begin by setting x_1 and x_2 to 0, making the origin the first feasible solution. In more complicated cases, the origin may not lie in the feasible solution space, and the first phase of the algorithm entails finding a first feasible solution. Since a feasible solution is not guaranteed to exist, the first phase is in effect an algorithm for determining the satisfiability of the system of inequalities. (This part of the simplex algorithm is used in CLP($\Re$), a constraint satisfaction system to be discussed in Section 4.2.)

The simplex algorithm saves computation by considering only those intersection points which can yield a solution better than the one just computed. By considering the gradient vector of the objective function at each iteration (i.e. the increase or decrease of the function with respect to each variable), the algorithm determines which variable to include in the set of basis variables next.

Details of the simplex algorithm can be found in any standard text on optimization (e.g. [Hestenes 1975]).

3.3. Numerical Constraint Satisfaction Systems

3.3.1. General Equation Solvers

Equation solving techniques used in symbolic algebra systems such as MACSYMA [Moses 1971] and Mathematica [Wolfram 1989] are extremely powerful in that they can solve systems of linear equations. Unfortunately, their generality leads to complex algorithms. Our need for fast algorithms to perform constraint satisfaction leads us to investigate alternative systems for algebraic simplification and equation-solving. Specifically, this section will discuss the application of term rewriting systems based on equational logic.

Two term rewriting systems that have been used in constraint programming are the Purdue Equational Interpreter [O'Donnell 1985] and Bertrand [Leler 1988]. We will concentrate on Bertrand, a language that was specifically designed to create constraint systems for numerical problems.

Bertrand is a rule-based specification language which lends itself to the building of constraint satisfaction systems. In Bertrand, the user can specify a system of constraints as a set of rules. These rules are interpreted by an extension of term rewriting called *augmented term rewriting*. To standard term rewriting, Leler has added the ability to bind values to variables. As a subject expression is rewritten and variables receive values, constraints are satisfied.

In this scheme, a constraint satisfaction system is comprised of three parts:

--a set of rewrite rules,

--a subject expression (the *main* of the program), and

--an augmented term rewriting system (the control mechanism).

A rule consists of a head and a body.

X + 0	*{X}*
head	*body*

A *subject expression* can be rewritten if it contains a subexpression matching the head of some rule. For example, the subject expression

*((8 - 2) + 0) * 9*

contains the subexpression *(8-2) + 0,* which matches *X + 0.* In the head of the rule, *X* is a parameter variable which can match an arbitrary expression. The matching process substitutes *(8-2)* for *X*, and the subject expression is rewritten to *(8-2) * 9.*

Binding is accomplished with a special infix operator *is*. The left argument of *is* must be a bindable variable; the right argument can be an arbitrary expression. For example, the expression $x = y + 5$ can be rewritten as x *is* $y + 5$. The *is* operator has important side-effects: It binds its right argument as the value of its left argument, and all other occurrences in the subject expression of the newly-bound variable name are replaced by the variable's value.

For example, the subject expression

$x = y + 5; x = y * 2$

can be rewritten to

x *is* $y + 5; y + 5 = y * 2.$

If a library of rules for algebraic simplification is included, the remaining expression can be solved for *y*. Thus, a system for handling numeric constraints, including those containing cycles, can be written in Bertrand and solved by means of augmented term rewriting.

It is also possible to define structured objects in Bertrand; that is, Bertrand allows the programmer to model an object by a part-subpart decomposition. A primitive object is created by a rule such as

aNumber {true}.

This rule will match a subexpression containing the nullary operator *aNumber*. As a side-effect, when the labeled expression

n: aNumber

is rewritten, an object named *n* of type rational number is created in the name space.

Higher-level objects can be built from the primitives. For example, the rule

aPoint {x: aNumber; y: aNumber; true}

can be applied to the subject expression *p1: aPoint*, resulting in the creation of subparts *p1.x* and *p1.y*.

By the same mechanism, constraints can be placed on the parts of a structured object. We can create an instance of a horizontal line with the following rewrite rule:

aHorizLine {p: aPoint; q: aPoint; p.y = q.y}

Finally, a typing mechanism makes it possible to overload operators. For example, if a "horizontal" operator were defined for both lines and squares, the rule could be "typed" as follows:

horiz L'line {L.p.y = L.q.y}

The 'line indicates that the *horiz* operator in the rule will match only parameters which are lines. To complete the typing, a type name must be attached to an object created by a rule, as in

aLine {p: aPoint; q: aPoint; true} 'line.

While Bertrand does not have interactive graphics capability, graphic output can be achieved with a number of primitive operators for drawing lines, circles, rectangles, and character strings. Graphical objects are not drawn automatically, but a postfix *!* (bang) operator is supplied by the system. A bang applied to a graphical object will cause the object to be drawn. Thus, a "midpointline" object can be defined as shown in Listing 2. (This object has become the canonical example for another constraint satisfaction system, ThingLab, to be discussed in Section 3.3.2.2).

```
aMidpointline {     line: aLine;
                    mid: aPoint;
...center constraints
          mid.x = (line.end.x + line.begin.x)/2;
          mid.y = (line.end.y + line.begin.y)/2
   } 'midpointline
...draw command
mpl' midpointline ! { mpl.line!; mpl.mid!; true}
main {
.. a simple test of the midpoint constraint
          mpline: aMidpointline;
          mpline.line.begin.x = 0.5;
          mpline.line.begin.y = 0.5;
          mpline.line.end.x = 2.5;
          mpline.line.end.y = 2.5;
          mpline !
}
```

Listing 2. Bertrand code for midpointline

A more interesting example (because it entails solving a system of linear equations) is a model of an electrical circuit. The circuit connects two 100 ohm resistors in series with a 10 volt battery. The program shown in Listing 3 outputs the current for this circuit.

```
resistance' linear resistor {
        voltagein: aNumber;
        voltageout: aNumber;
        current: aNumber;
        voltagein - voltageout = current * resistance
  } 'eob

voltage' linear battery {
        voltagein: aNumber;
        voltageout: aNumber;
        current: aNumber;
        -voltage = voltagein - voltageout
  } 'eob

a' eob series b' eob {
        a.current = b.current &
            a.voltageout = b.voltagein
  }

n kilo { 1000 * n}
n volt {n}
n ohm {n}
ground {0}

main {
        b1: (10 volt battery);
        r1: (100 ohm resistor);
        r2: (100 ohm resistor);
        b1 series r1;
        r1 series r2;
        r2 series b1;
        b1.voltagein = ground;
        r1.current
  }
```

Listing 3. Electrical circuit in Bertrand

The main of this program drives the entire process. Its first statement *b1: (10 volt battery)* results in a series of rewrites. *10 volt* matches the head of *n volt {n}* and is thus rewritten as *10*. *10 battery* matches *voltage' linear battery*, since *10* is an instance of *linear*. This then results in a new object being placed in the symbol table. This object, named *b1*, has parts *voltagein*, *voltageout*, and *current*, and the constraint that

-10 = b1.voltagein - b1.voltageout.

Similar rewriting creates resistor objects *r1* and *r2* with the added constraints

*r1.voltagein - r1.voltageout = r1.current * 100*
*r2.voltagein - r2.voltageout = r2.current * 100.*

The next three statements in main specify that *b1* is in series with *r1*; *r1* is in series with *r2*; and *r2* is in series with *b1*. The rewriting rule associated with *series* adds the following constraints:

b1.current = r1.current
b1.voltageout = r1.voltagein
r1.current = r2.current
r1.voltageout = r2.voltagein
r2.current = b1.current
r2.voltageout = b1.voltagein.

The next statement adds the constraint

b1.voltagein = 0

since *ground* rewrites to *0*. The final statement in main returns the value of *r1*'s subpart *current*. This value (*current = 0.05*) is computed by solving the simultaneous linear equations represented by the above constraints, a process which can be handled by additional rewriting rules.

Bertrand's primary strength is its facilities for building new constraint systems. Its weaknesses are speed and the fact that the augmented term rewriting system cannot handle inequalities, nor can it produce multiple solutions to a problem. These features are offered by Prolog III and CLP($\Re$), the constraint satisfaction systems discussed in Section 4.

3.3.2. Constraints for Graphical Applications

Constraint-based programming lends itself readily to graphics applications. Because a geometric object is defined by the relative sizes and positions of its parts, it is usually easier to describe the object declaratively rather than procedurally. Furthermore, the bi-directionality of a constraint-based description facilitates user-interface construction, allowing the user to manipulate the object while the constraint satisfaction system ensures that the object keeps its essential character based upon the constraint description.

3.3.2.1. Sketchpad

Ivan Sutherland's Sketchpad was the first system to take advantage of constraint satisfaction for the definition and maintenance of geometric objects [Sutherland 1963].

Sketchpad is an interactive system in which the user can sketch a rough version of a geometric figure, add constraints to it, and create a primitive from the object with the *macro* facility. Constraints are limited to a pre-defined primitive set, including

--making two lines parallel, perpendicular, or of equal length;

--making a line horizontal or vertical;

--constraining a point to lie on a line or arc; and

--producing digits on the display for some scalar value.

Figure 15 shows how an equilateral hexagon can be created using constraints that force the hexagon's vertices to lie on a circle and then using additional constraints that force its sides to be of equal length.

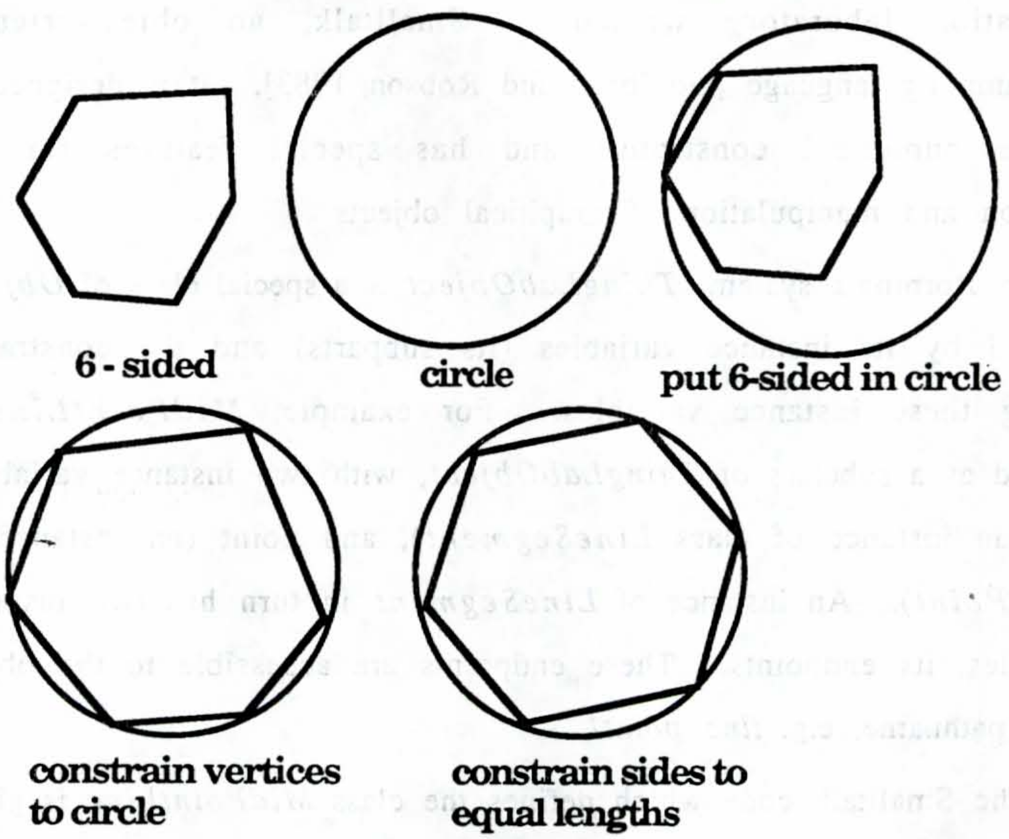

6 - sided
circle
put 6-sided in circle
constrain vertices to circle
constrain sides to equal lengths

Figure 15. Using Sketchpad constraints to create an equilateral hexagon

Merge constraints, performed recursively on the subparts of an object, are used to replace two objects with a single equivalent object. Constraint satisfaction is accomplished using propagation of degrees of freedom and relaxation for constraint sets involving cycles. Relaxation takes its initial values from the user's rough sketch of an object, making it faster.

Sketchpad was ahead of its time in its use of constraints for interactive graphics, and it served as a model for later systems such as ThingLab.

3.3.2.2. ThingLab

ThingLab [Borning 1979; 1981; 1986] is a constraint-based simulation laboratory written in Smalltalk, an object-oriented programming language [Goldberg and Robson 1983]. It is designed to handle numerical constraints and has special features for the creation and manipulation of graphical objects.

In Borning's system, *ThingLabObject* is a special class of *Object*, defined by its instance variables (its subparts) and the constraints among these instance variables. For example, *MidPointLine* is defined as a subclass of *ThingLabObject*, with two instance variables: line (an instance of class *LineSegment*), and point (an instance of class *Point*). An instance of *LineSegment* in turn has two instance variables, its endpoints. These endpoints are accessible to the object via a pathname, e.g. *line point1*.

The Smalltalk code which defines the class *MidPointLine* is given in Listing 4.

There are two constraints defined on the object. The first requires that the midpoint be halfway between the endpoints. Notice that because ThingLab has no facilities for algebraic manipulation, the user must supply local methods for maintaining the constraint. In this example there are three such methods, one for computing each point, given the other two.

```
ThingLabObject subclass: #MidPointLine
      instanceVariableNames: 'line point'
      classVariableNames: ''
      poolDictionaries: ''
      category: 'Prototypes'.

MidPointLine prototype parts: 'line'.
MidPointLine prototype locations: 'point'.
MidPointLine prototype field: 'line'
      replaceWith: (40@40 line: 100@20).
MidPointLine prototype field: 'point'
      replaceWith: 70@30.

MidPointLine prototype inserters: #('line point1' 'line point2').
MidPointLine prototype constrainers: #('line').

(Constraint owner: MidPointLine prototype
      rule: 'point = ((line point1 + line point2) // 2)'
      error: 'line location dist: point'
      methods: #(
        'self primitiveSet.point: (line point1 + line point1)//2'
        'line primitiveSet.point2: line point1 + (point - line point1* 2)'
        'line primitiveSet.point1: line point2 + (point - line point2 * 2)')
      priority: #required) isOKtoSplit.

Constraint owner: MidPointLine prototype
      stay: #('point')
      priority: #weakDefault.
```

Listing 4. Smalltalk code for Thinglab midpointline

ThingLab provides an interactive interface, the ThingLabBrowser, where the user can define and edit graphical objects (Figure 16). A number of pre-defined graphical objects are supplied, with which the user can build more complex objects without having to resort to Smalltalk code. Objects constructed in this manner inherit constraints from their subparts.

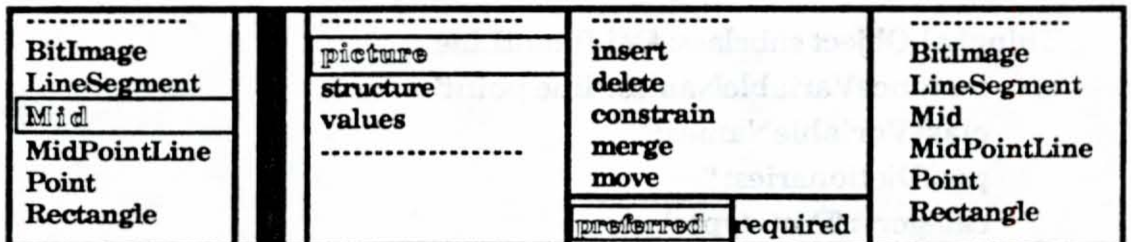


Figure 16. ThingLab's midpointline

The *ThingLabBrowser* offers three views of an object: its picture, its structure, and its values. The picture of a *ThingLabObject* reflects the current values of the object's prototype. With a mouse, the user can edit the prototypical object. For example, the user can pull on the endpoint of a *MidPointLine*, initiating constraint satisfaction. That is, the object tries to maintain its constraints at the same time that it satisfies the user's request.

For constraint satisfaction, *ThingLab* uses propagation of degrees of freedom and relaxation where the former fails. The algorithm begins by gathering all the constraints on the object and arranging them in a Constraint Hierarchy Graph. The ability to prioritize constraints distinguishes Borning's system from other numerical constraint satisfaction systems [Borning 1987; 1989].

A constraint can be defined at one of five levels:

C_0 -- Required (constraints which must be upheld)
C_1 -- Strongly Preferred (*anchors* for values which should not be changed during constraint satisfaction)
C_2 -- Preferred (user editing requests)
C_3 -- Default (*stay* constraints, which keep prototype values in place, unless a higher level constraint demands a change)
C_4 -- Weak default (*stays* with derived values, e.g. the midpoint of a *MidPointLine*)

When ThingLabObjects are defined, a *stay* is automatically placed on each object. The *weak default* on the midpoint in effect says that if one of the endpoints is moved, the user would prefer to move the midpoint in satisfaction of the required constraint, rather than move the other endpoint. This constraint is intended to be in keeping with *The Principle of Least Astonishment.* With the constraint hierarchy, an object-designer has a clear way of specifying how an object should change when constraints can be satisfied in more than one manner. The constraint to keep an endpoint in place is made stronger than the constraint to keep the midpoint in place because, when pulling on one endpoint, the user probably does not expect to see the other endpoint stretch out in the opposite direction.

Once the Constraint Hierarchy Graph has been constructed, the algorithm repeatedly traverses the graph from the C_0 level down, looking for a useable constraint. A constraint is useable if

-- it has a method associated with it for which there is sufficient input to produce an output; and

-- a change in the output of the candidate method will not violate a higher level constraint.

Typically, prototype values are gathered from the stay constraints in the first few passes, at which point there is sufficient information to derive output from a more interesting constraint.

The constraint satisfier finds useable methods from among those given in the definitions of the constraints and orders these methods such that values are propagated from one to another. The sequence of local methods constitutes a global method for satisfying the user request and maintaining the constraints on the object. This constraint satisfaction method is then compiled and placed in the object's method dictionary so that, if in the future the user makes a similar editing request, the object will know how to respond to it immediately.

3.3.2.3. ThingLab II

ThingLab's rapid response to an editing operation for which it has already compiled a constraint satisfaction method is in sharp contrast to its slow response after any changes are made to the constraints governing an object's behavior. Any change, no matter how simple, results in ThingLab's throwing out all methods that it has developed for that object. The consequence is that it must now create completely new methods, even if the changes have no effect on some of those methods already compiled. This slow operation during development makes ThingLab of limited use as a tool for producing interactive interfaces.

ThingLab II [Maloney, Borning, and Freeman-Benson 1989; Freeman-Benson, Maloney, and Borning 1990] has been designed as a constraint environment that is hospitable for the creation of interactive interfaces. It includes most, but not all of ThingLab's power, and yet it rapidly resolves constraint systems, even while the underlying constraints are undergoing constant alteration. The secret to ThingLab II's success is *incremental* constraint satisfaction.

The incremental method of satisfying constraints in ThingLab II is based on a deceptively simple concept called the *walkabout strength* of a variable. The walkabout strength of some variable *v* is the strength of the weakest constraint in the current solution that needs to be violated in order for *v* to have a value determined by some different constraint. Using walkabout strengths to guide its search, the incremental algorithm achieves the effect of a new constraint by searching for either an unconstrained variable or a constraint whose strength is sufficiently weak to allow it to be violated in order to satisfy the new, stronger constraint. While the search may visit all constraints in an effort to find one that can be retracted, experimental evidence indicates that this situation is rare. Changes to a constraint graph usually exhibit the *locality principle* commonly encountered in other areas of computer science.

While incremental constraint algorithms are mandatory for the rapid resolution of dynamic constraint systems, they are not necessarily best when we are faced with systems that rarely change. Their first and most glaring deficiency comes from the fact that these algorithms rarely work on constraint graphs that contain cycles. A second, more easily avoided problem is that they are inefficient when applied to a completely unresolved constraint system

Other derivative works from ThingLab include the Filters Project [Ege, Maier, and Borning 1987; Ege 1989] and Animus [Duisberg 1986]. The first of these concerns the development of user interfaces. The second adds animation based on temporal constraints.

3.3.2.4. The Filters Project

The Filters Project [Ege, Maier, and Borning 1987] grew out of the observation that a declarative specification of the model-view relationship might facilitate the construction of user interfaces. Smalltalk, the language in which ThingLab is written, relies on its Model-View-Controller paradigm [Goldberg and Robson 1983], an interface-building strategy which is more clear-cut in theory than in application. Ege's claim is that constraints provide a clearer, more natural description of the model-view relationship.

The Filters Project attempted to use Smalltalk's class object-oriented paradigm and ThingLab's constraint satisfaction system as a basis for a constraint filter specification language. A *filter* is an object which, in terms of constraints, describes the relationship between a source (the data model) and the user's view of it. It is constructed from *subfilters* relating subparts of the source and view. Subfilters can be placed together in a *filter pack* with *sequence*, *iteration*, and *condition* operators. A number of atomic filters (primitives) were provided by the implementation, including IntegerEquality, StringRender, StringConversion, and PopUpMenu.

The Filter Specification Language attempted by Ege was limited in its ability to express conditional and set constraints, which are necessary for a complete interface-construction package. For this reason, a new implementation is being attempted based upon the CLP($\Re$) constraint logic programming language [Ege 1989] (Section 4.2). Issues of part-subpart and class-subclass inheritance are being addressed in this new version.

3.3.2.5. Animus

Animus is a ThingLab-inspired system which uses constraints for animated user interfaces [Duisberg 1986]. The Animus system adds three classes of temporal constraints to ThingLab's static constraints: *TimeDifferentialConstraint*, *TimeFunctionConstraint*, and *TriggerConstraint*. The first class handles simulations of continuous time relations, such as $i = dq/dt$; the second class handles values which are a function of time, such as *verticalOffset* = $A \sin wt$; and the third class handles actions which are triggered by discrete events, such as objects moving across the display screen in response to certain messages.

The name *Animus* refers to the idea that each animated object has an anima or soul that gives it life and sets it in motion. When a *ThingLabObject* is created (e.g. *Circuit*), the system automatically creates a corresponding anima class (*CircuitAnima*) and its prototype instance. All such anima classes are subclasses of *Anima*, they hold instance variables for *time* and *eventQueue*, and they manage constraint satisfaction for the animated object.

Temporal constraints are defined on instances of *ThingLabObject* in ThingLab style. An object may have both static and temporal constraints on it. For example, a dynamic electrical circuit might contain a resistor with the constraint

*delta v = i * r*

as well as an inductor with the constraint

delta v/L = di/dt.

However, when the *ThingLabObject* is instantiated, all temporal constraints migrate to the object's anima, since it is the anima which responds to the tick of the global clock and keeps track of the event queue.

A temporal constraint is a relation between a stimulus (e.g., a tick of the clock or the receipt of some message) and a response in the form of an event or a stream of events. An instance of class *Event* is a representation of a Smalltalk method along with a time stamp specifying at what time the event-message is to be sent.

As the simulation begins, the anima compiles responses for all of its temporal constraints. The compiled methods contain messages which will place the constraint's response events on the event queue when the constraint is triggered at run-time. The events are ordered by time stamp so that they can be activated at the appropriate moment. The time stamps also make it possible to interleave animations of different objects so that they appear simultaneous.

In the case of a *TimeDifferentialConstraint*, the constraint satisfaction method is fairly simple. Consider an electrical circuit containing a capacitor with the constraint $i = dq/dt$. This constraint requires that the charge on the capacitor be adjusted upon each tick of the clock. Constraint satisfaction yields a finite difference approximation of the differential equation by assuming that the value for i at the onset of each discrete time interval is good for the duration of that interval. Thus, the message which increments i is placed on the event queue and is sent at the next tick of the clock.

A *TimeDifferentialConstraint* is satisfied with an *ImmediateResponse*, that is, a single event which occurs at the next time instant. *TimeFunctionConstraints* are handled similarly.

TriggerConstraints, on the other hand, generally initiate more elaborate responses, instances of class *Flasher, Trajectory, Script*, or *Sequence*. These responses are comprised of a series of time-stamped events which are placed on the event queue. A *Sequence*, for example, points to a file containing an array of bitmaps to be displayed one after the other. A *Trajectory* response causes an object to move across the display screen at a given velocity or following a given path. A *Flasher* specifies a rectangle on the screen which flashes from black to white.

Below is an example of a *TriggerConstraint* placed on an instance of SortQueue.

```
TriggerConstraint owner: SortQueue prototype
    to: 'list'
    trigger: #compare:with:
    causes:     '(Flasher with: (list at: a1)) enqueueEvents.
                 (Flasher with: (list at: a2)) enqueueEvents'
```

The list is an instance variable of *SortQueue* and contains the values to be sorted. The constraint states that when the list receives the message *compare:with:*, a series of events is to be placed on the event queue causing the graphical display of the values being sorted to flash. The constraint causes the *SortQueue*'s anima to compile the method given below:

```
thingsortQueue3list.compare: t1 with: t2
    (Flasher with: (thing sortQueue3 list at: t1)) enqueueEvents.
    (Flasher with: (thing sortQueue3 list at: t2)) enqueueEvents.
    self tick: 1.
    ↑thing sortQueue3 list compare: t1 with: t2.
```

Then all calls to *compare:with:* are replaced by calls to this newly-compiled method. The intention in this design is to separate the application program from the code implementing its animation.

3.3.2.6. METAFONT

METAFONT uses constraints and an equation solver to describe the shape of a character of text [Knuth 1979]. The constraints are restricted to sets of linear equations that can be solved in one pass. The system can be run interactively, the user specifying the positions of points by constraints and drawing the character a command at a time with the points as parameters. However, the user cannot edit the resulting image by pointing to it with a mouse, a limitation addressed by a later system, JUNO.

3.3.2.7. JUNO

JUNO is a constraint-based graphics system which adds a what-you-see-is-what-you-get editor to a METAFONT-style language [Nelson 1985]. Like METAFONT, JUNO allows the user to describe an object by geometric constraints, but constraints are not limited to linear equations, allowing for geometric predicates of parallelism and congruence.

Another added feature is the ability to select a pre-defined constraint, represented graphically by an icon, and apply it to a point with the mouse. While this simplifies the definition of constraints, it limits the applications since the only data object is a point, and only four constraints are available: Two points can be constrained to be horizontal or vertical to each other, and four points can be constrained to describe parallel or congruent lines.

One interesting feature of JUNO is the facility for building an image both textually and graphically. An image produced by a constraint specification can be moved or stretched with the mouse, and the underlying constraint description is changed accordingly.

The JUNO constraint solver uses Newton-Raphson iteration, a derivative-based method faster than relaxation. Like Sketchpad's relaxation technique, this method benefits from hints from the user, who must lay out the points of a graphical object in roughly the right position in order to give the iteration process a headstart.

3.3.2.8. IDEAL

IDEAL is a high-level graphics language which can be used to typeset figures so that they can be printed on the same page with text, eliminating the need to cut and paste [Van Wyk 1982].

IDEAL's procedures, called *boxes*, describe a graphical object by means of:

-- a set of constraints on the relative positions of points;

-- a set of drawing commands; and

-- a list of boundary points.

For example, the rectangle box is defined as:

```
        rect {
                var ne, nw, sw, se, c, ht, wd;
                ne = se + (0,1)*ht;
                nw = sw + (0,1)*ht;
                ne = nw + wd;
                c = (ne + sw)/2;
                conn ne to nw to sw to se to ne;
        }.
```

Complex numbers are used to describe points because they can implicitly express operations such as translation or rotation.

A call to a box is made with a *put* statement, accompanied by enough parameters and/or additional constraints to uniquely describe the object. For example, both of the following calls draw the same rectangle:

```
put rect {
        ht = 2;
        wd = 1;
        sw = 0;
}

put rect {
        ht = 2;
        wd = 0.5 * ht;
        nw = (0, 2);
}.
```

IDEAL has procedures for drawing rectangles, circles, and arcs. It also has facilities for drawing an object iteratively (*pens*), obscuring one object with another, and applying a texture over a polygonal area.

Constraints must be expressed as linear or slightly non-linear equations. IDEAL's equation solver is similar to METAFONT's, but more powerful in one respect: If necessary, it will search for a proper ordering of the equations such that the sequence of substitutions transforms the equations from non-linear to linear. In METAFONT, the equations are processed in the order given, and no other orderings of substitutions are tried.

3.3.3. Constraints for Electrical Circuits

Constraints can also be applied to the modeling of electrical circuits, as illustrated by Sussman and Steele's constraint satisfaction system [Sussman and Steele 1979]. Their constraint language interpreter is built around a number of primitive constraints, including *adders* and *multipliers*. The constraints have subparts, down to primitive cells (e.g. numbers), which can be accessed by a pathname (e.g. *adder1 a1*). Simple constraints can be connected into arbitrarily complicated networks with a *merge*.

This constraint language is like ThingLab in its hierarchical construction of objects. Figure 17 gives the definition of a resistor and the corresponding schematic diagram. Pathnames are preceded by **>>**, and merges are denoted by **==**.

An advantage of this system is that it keeps track of dependencies and can tell the user how conclusions were drawn. A premise can be changed, and all conclusions dependent on it are automatically retracted.

The primary constraint satisfaction technique used is propagation of known states. If propagation fails due to cycles in the constraint network, another strategy can be used appropriate for the electrical circuit application, namely, equivalent views.

It is sometimes possible to break a cycle in a constraint network by offering an alternate view. For example, the constraint

$$X + X = 4$$

has a cycle, while

$$2X = 4$$

does not.

When translated into redundant views in a constraint network, equivalent views of a circuit can lead to a design solution. For example, the system can be given the additional knowledge that two resistors in series are equivalent to a single resistor.

A RESISTOR

```
(constraint resistor
    ((t1 terminal)
    (t2 terminal)
    (resistance number)
    (av adder)
    (ai adder)
    (m multiplier))
    (==(>> v t1) (>> sum av))
    (== (>> v t2) (>> a1 av))
    (== (>> a2 av) (>> product m))
    (== (>> i t1) (>> m1 m))
    (== resistance (>> m2 m))
    (== (>> i t1) (>> a1 ai))
    (== (>> i t2) (>> a2 ai))
    (constant (>> sum ai) 0.0))
```

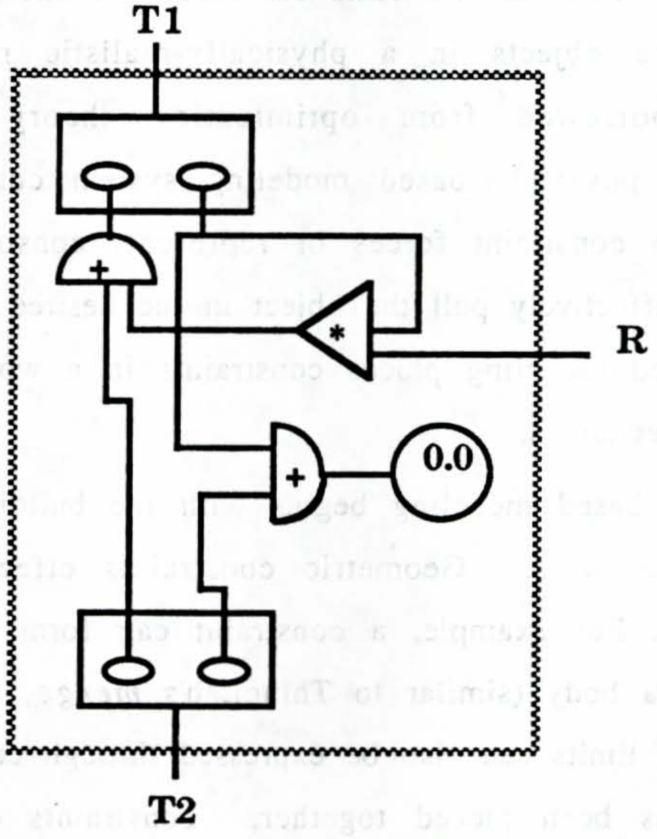


Figure 17. Sussman and Steele resistor constraints

Sussman and Steele call the equivalent views *SLICES*, and they use them as an alternative to relaxation for breaking cycles. They also discuss, but do not implement, techniques for algebraic manipulation that would make their system more powerful.

3.3.4. Constraints for Physically-Based Modeling

In the systems and applications discussed so far, constraints are viewed as mathematical relations which can be "solved" in more or less direct ways. For example, we get new values representing an object's state by propagating changes, or we solve the set of constraints as a system of linear equations.

Physically-based modeling, a growing area of graphics research, interprets constraints as the basis for forces or energies that can be used to move objects in a physically-realistic manner. Using techniques borrowed from optimization theory and classical mechanics, a physically-based modeling system converts constraint equations into constraint forces or represents constraints as energy fields which effectively pull the object in the desired direction. Thus physically-based modeling places constraints in a world governed by Newtonian mechanics.

Physically-based modeling begins with the building of an object from primitive parts. Geometric constraints effectively snap the parts together. For example, a constraint can form a joint between two parts of a body (similar to ThingLab's *merge*, which joins two points). Joint limits can also be expressed through constraints. Once the object has been pieced together, constraints can be used to position or animate it. For example, a constraint can anchor a point to a location in space, or it can require a point to follow a predefined path [Barzel and Barr 1988].

The two most commonly-reported constraint methods for physically-based modeling are reviewed in the next two sections. These methods have been adapted and augmented by a number of researchers, as discussed in Section 3.3.4.3.

3.3.4.1. Penalty Method

The penalty method is a technique borrowed from optimization theory. The basic optimization problem is to find a vector $\overline{x}$ which locally optimizes (minimizes or maximizes) a function $f(\overline{x})$. The basic constrained optimization procedure optimizes the function subject to a given constraint. For example, we may wish to

minimize $f(x_1, x_2) = 2x_1^2 + 3x_2^2$

subject to $g(x_1,x_2) = x_1 + 3x_2 - 10 = 0,$

where $f(x_1,x_2)$ is the objective function and $g(x_1,x_2) = 0$ is the constraint.

The penalty method looks for an approximate solution to this problem by first defining a new objective function $h(\overline{x})$ which includes a penalty for violating the constraint. Since we want the penalty to indicate how far we are from satisfying the constraint, it is reasonable that the penalty should be of the form $k[g(\overline{x})]^2$. Now the problem is to minimize

$$h(\overline{x}) = f(\overline{x}) + k[g(\overline{x})]^2.$$

The non-negative constant k is a weighting factor which indicates how strongly we insist on the constraint. When $k = 0$, the constraint is ignored. As $k \rightarrow \infty$, the constraint is completely satisfied. Figure 18 illustrates how $h(\overline{x})$ approaches the constrained minimum for increasing values of k.

The penalty formulation of a minimization problem can be used to model the constrained motion of an object. Here, the objective function $U(\overline{x})$ represents the potential energy of the unconstrained physical system, while the penalty function $k[g(\overline{x})]^2$ acts like a spring with its own potential energy, attempting to pull the object in the direction of the constraint:

$$E(\overline{x}) = U(\overline{x}) + k[g(\overline{x})]^2 \qquad [1]$$

The penalty method is also easily generalized to multiple constraints since the constraint terms can be summed. The problem

minimize $U(\overline{x})$, subject to $g_\alpha(\overline{x}) = 0;\ \alpha = 1, 2, \ldots, n$

becomes

$$\text{minimize } E(\overline{x}) = U(\overline{x}) + \sum_{\alpha=1}^{n} k_\alpha [g_\alpha(\overline{x})]^2 \qquad [2]$$

Since the object will minimize its potential energy over time, we can now solve the constrained motion problem as a minimization problem. The simplest optimization algorithm is a numerical procedure called ***gradient ascent/descent***. (We will put aside our energy function for the moment and illustrate the gradient ascent (***hillclimbing***) procedure with a two-dimensional function since this is easier to depict graphically. We will then return to solve the energy function for our physical model.)

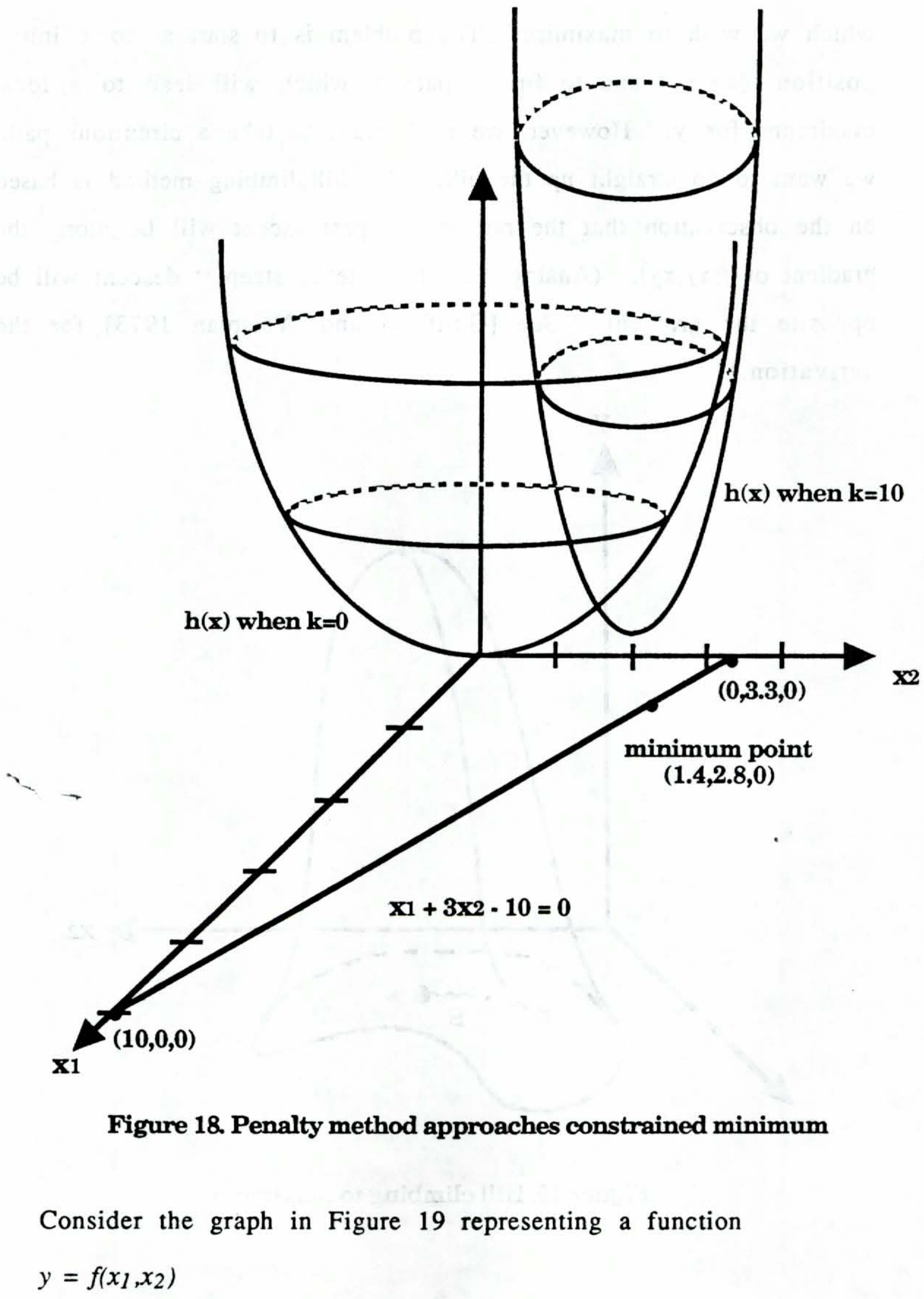


Figure 18. Penalty method approaches constrained minimum

Consider the graph in Figure 19 representing a function

$y = f(x_1, x_2)$

which we wish to maximize. The problem is to start at some initial position (x_1,x_2) and to find a path S which will lead to a local maximum for y. However, we don't want to take a circuitous path; we want to go straight up the hill. The hillclimbing method is based on the observation that the rate of steepest ascent will be along the gradient of $f(x_1,x_2)$. (Analogously, the rate of steepest descent will be opposite the gradient. See [Gottfried and Weisman 1973] for the derivation.)

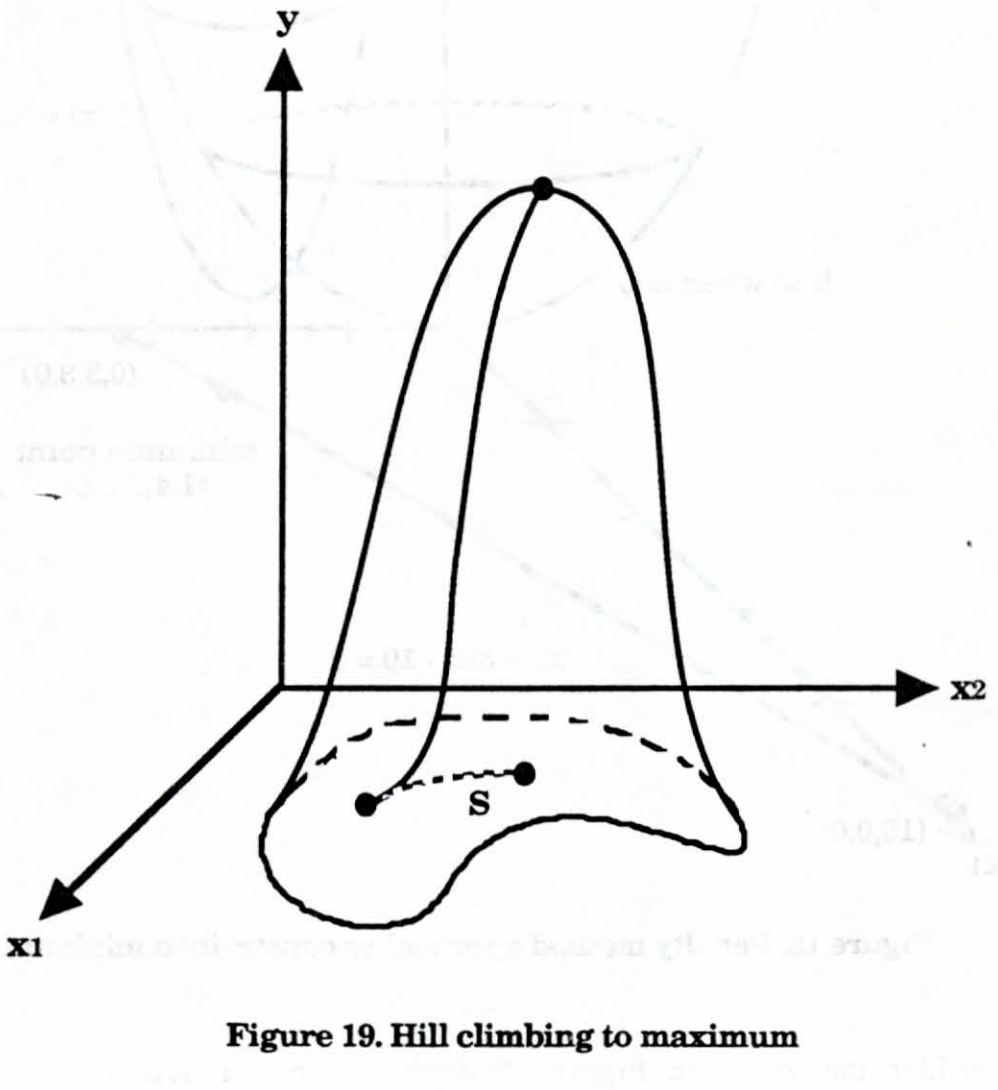


Figure 19. Hill climbing to maximum

We can understand intuitively why the hillclimbing method works if we picture the contour lines of the objective function projected down to the x_1, x_2 plane (Figure 20). (The contour lines represent the contours of the function when y is set to different constants.) Saying that we want to move in the direction of the gradient of the function is equivalent to saying that we want our path S always to run perpendicular to these contour lines. This will take us directly up the "hill" as the values of x_1 and x_2 vary. Thus, we have:

$$\frac{dx_i}{dt} = \frac{\partial f}{\partial x_i} \quad \text{for } i = 1,2.$$

(For steepest descent, the variables move in the opposite direction of the gradient: $\frac{dx_i}{dt} = -\frac{\partial f}{\partial x_i}$).

We can now use Euler's method to solve the differential equations. Let x_{ik} denote the value of independent variable x_i at time t_k. To get the value of the independent variables at time t_{k+1}, we use:

$$x_{ik+1} = x_{ik} + h\frac{\partial f}{\partial x_i} ,$$

where h is the step size. We continue to step up the hill in this manner until the gradient becomes sufficiently small to indicate that we have arrived at a maximum.

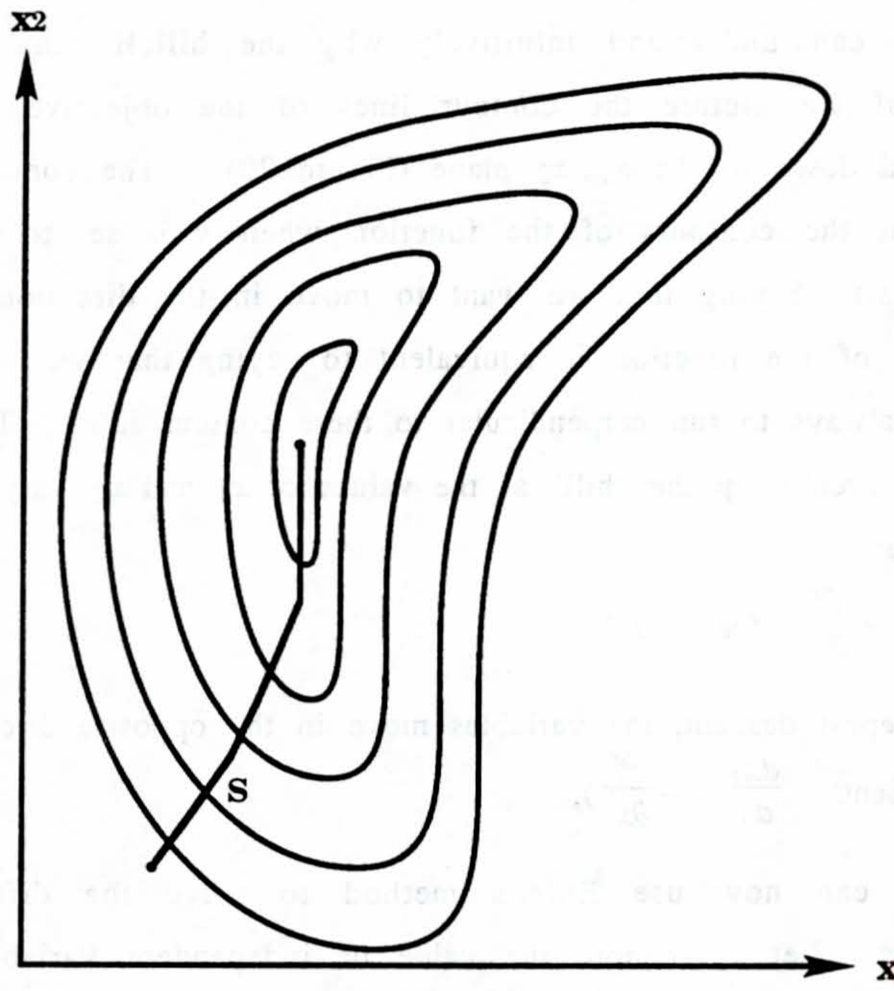


Figure 20. Projection of contour lines onto x1-x2 plane

As noted above, the penalty method can be used to minimize the potential energy of a physical object which we wish to place in motion. The objective function to be minimized is given by equation [1]. Applying the gradient descent method, we move the independent variables in $\overline{x}$ in a direction opposite to the gradient of the function. This yields:

$$\Sigma_j M_{ij} \frac{dx_j}{dt} = F(\overline{x}) - 2kg(\overline{x}) \frac{\partial g}{\partial x_i} \qquad [3]$$

where M_{ij} is the generalized mass matrix and $F(\overline{x})$ is the generalized force on the system [Platt 1989].

A simple example (Figure 21) will illustrate this method. Let (x_1,x_2), (x_3,x_4) denote the positions of two unit mass balls in 2-d space. Their initial positions are (0.4,0) and (0.5,0), respectively. The balls are constrained by "springs" to the (0,1) and (1,1) positions, respectively, and they are also attached to each other by a spring.

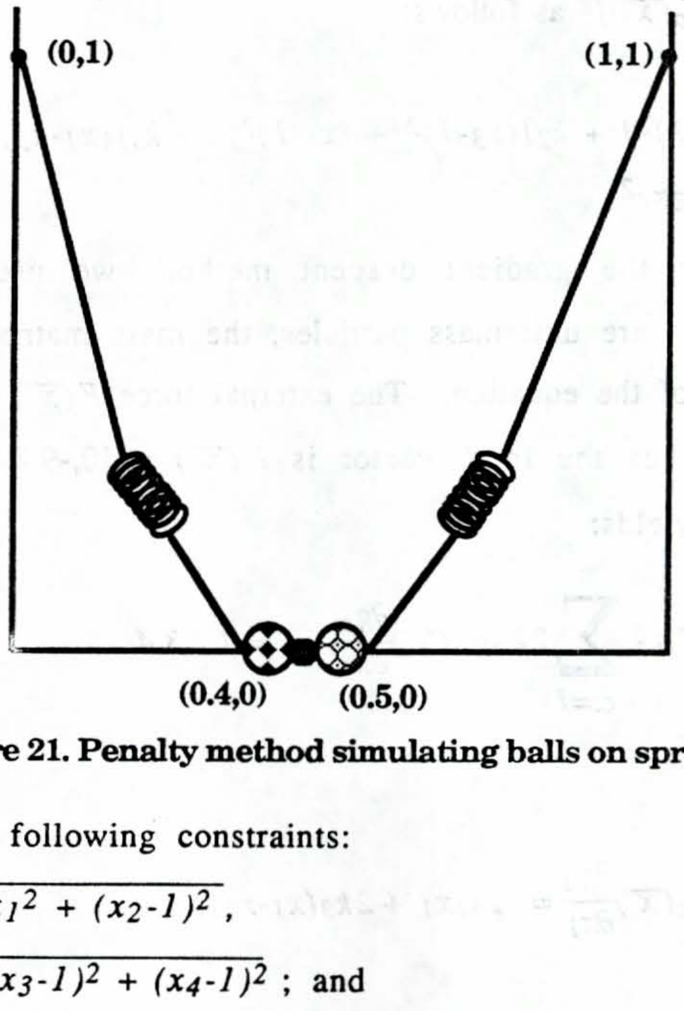


Figure 21. Penalty method simulating balls on springs

This gives the following constraints:

$$g_1(\overline{x}) = \sqrt{x_1^2 + (x_2-1)^2}\ ,$$

$$g_2(\overline{x}) = \sqrt{(x_3-1)^2 + (x_4-1)^2}\ ;\ \text{and}$$

$$g_3(\overline{x}) = \sqrt{(x_1 - x_3)^2 + (x_2 - x_4)^2}\ .$$

The balls are also constrained not to go below the ground. To achieve this effect, we activate a penalty force only when a ball moves to a negative y position, as follows:

$g_4(\overline{x}) = u_1 x_2$ where $u_1 = 1$ when $x_2 < 0$; else $u_1 = 0$;

$g_5(\overline{x}) = u_2 x_4$ where $u_2 = 1$ when $x_4 < 0$; else $u_2 = 0$.

Giving each constraint $g_\alpha(\overline{x})$ a weight of k_α, we get a penalty term $\sum_{\alpha=1}^{n} k_\alpha [g_\alpha(\overline{x})]^2$ as follows:

$$k_1[x_1^2 + (x_2-1)^2] + k_2[(x_3-1)^2 + (x_4-1)^2] + k_3[(x_1-x_3)^2 + (x_2-x_4)^2] + k_4 u_1 x_2^2 + k_5 u_2 x_4^2$$

To apply the gradient descent method, we use equation [3]. Since the balls are unit mass particles, the mass matrix is the identity and falls out of the equation. The external force $F(\overline{x})$ consists only of gravity, and thus the force vector is $F(\overline{x}) = [0,-9.8,0,-9.8]$. Thus, equation [3] yields:

$$\frac{dx_i}{dt} = F(\overline{x})_i - \sum_{\alpha=1}^{n} 2k_\alpha g_\alpha(\overline{x}) \frac{\partial g_\alpha}{\partial x_i}, \quad i = 1,2,3,4$$

where

$$\sum_{a=1}^{n} 2k_a g_a(\overline{x}) \frac{\partial g_a}{\partial x_1} = 2k_1 x_1 + 2k_3(x_1-x_3);$$

$$\sum_{a=1}^{n} 2k_a g_a(\overline{x}) \frac{\partial g_a}{\partial x_2} = 2k_1(x_2-1) + 2k_3(x_2-x_4) + 2k_4 u_1 x_2;$$

$$\sum_{\alpha=1}^{n} 2k_\alpha g_\alpha(\overline{x})\frac{\partial g_\alpha}{\partial x_3} = 2k_2(x_3-1) - 2k_3(x_1-x_3); \text{ and}$$

$$\sum_{\alpha=1}^{n} 2k_\alpha g_\alpha(\overline{x})\frac{\partial g_\alpha}{\partial x_4} = 2k_2(x_4-1) - 2k_3(x_2-x_4) + 2k_5 u_2 x_4.$$

Integrating these with a time step of 0.01 and then sampling every tenth point results in motion of the balls characterized by the time and positions in Table 1.

Time	x1	x2	x3	x4
0.0	0.40	0.00	0.50	0.00
0.1	0.20	0.66	0.37	0.58
0.2	0.16	0.37	0.31	0.16
0.3	0.15	0.28	0.30	0.02
0.4	0.15	0.26	0.29	-0.01
0.5	0.14	0.28	0.29	0.02
0.6	0.14	0.29	0.29	0.05
0.7	0.14	0.31	0.29	0.10
0.8	0.14	0.26	0.29	0.00
0.9	0.14	0.26	0.29	0.00
1.0	0.14	0.28	0.29	0.03

Table 1. Positions of balls in penalty method

While the penalty method is fairly simple to use, it does not satisfy constraints exactly. This is an advantage in that a compromise among constraints is sometimes desirable, but a disadvantage in that an object held together by multiple "springs" may look too loosely connected. Another disadvantage is that as the constraint weights increase, the differential equations become *stiff* due to the widely separated time constants. Most numerical methods require time steps on the order of the fastest time constant. However, such large time steps may cause the "springs" to bounce unrealistically.

Application of the penalty method in physically-based modeling is discussed in [Witkin, Fleischer, and Barr 1987], [Platt and Barr 1988], and [Platt 1989].

3.3.4.2. Inverse Dynamics

In some applications, it is necessary that constraints be fulfilled exactly. This can be accomplished by a process of *inverse dynamics*. The forward dynamics problem entails computing an object's behavior given the forces which act on it. The inverse dynamics problem is just the opposite: Given the constraint equations which define an object's structure, location, and behavior, we compute *constraint forces* which cause the object to move in an appropriate manner. A constraint force works like an invisible hand which guides an object in the correct direction or prevents it from moving beyond its limits no matter what other external forces are exerted on the object.

Witkin, Gleicher, and Welch [1989] present a derivation of the physical equation which yields the appropriate constraint force. Given in the problem are a set of constraint functions $c_i(q, t)$ and a vector q of the object's independent variables, and a time t. Each constraint function c_i depends on the state of the independent variables and possibly also on time. We say that the constraint is satisfied when $c_i(q, t) = 0$.

The goal is to find a constraining force C which, when added to the known external force Q, will result in motion which is consistent with the constraints. The object will be moved in accordance with Newton's second law of motion, which in generalized form is:

$$M_{ij}\ddot{q}_j = C_j + Q_j. \qquad [1]$$

M is the generalized mass matrix of the object being moved.[2] C is the vector of unknown constraint forces, and Q is the vector of known external forces with respect to each independent variable. The constraint force effectively cancels any component of Q which would cause the object to violate its constraints. Once C is known, it can be added to Q. Then $\ddot{q}$, the second time derivative of the independent variables, can be determined. It is then possible to integrate the differential equation over time and move the object.

This equation cannot be solved directly since both C and $\ddot{q}$ are unknown. We need more information. Since we want the constraints always to be satisfied, we know that each constraint c_i must be 0 at initial time t_0, the rate of change of c_i must be 0, and that rate of change must not change from 0. Thus we have $\ddot{c}_i = 0, 1 \le i \le n$ (where n is the number of constraints).

Finding $\ddot{c}_i$ and substituting into $\ddot{q}_j = W_{jk} (C_k + Q_k)$, we get

$$\frac{\partial c_i}{\partial q_j} W_{jk} (C_k + Q_k) + \frac{\partial \dot{c}_i}{\partial q_j} \dot{q}_j + \frac{d^2 c_i}{dt^2} = 0 \qquad [2]$$

where W is the inverse of matrix M.

---

[2] According to the summation convention, the appearance of an index twice in a term implies summation. Thus $M_{ij}q_j$ is equivalent to $\Sigma_j M_{ij} q_j$, i.e., row i of matrix M times vector q.

Rearranging, we get:

$$-\frac{\partial c_i}{\partial q_j} W_{jk} C_k = \frac{\partial c_i}{\partial q_j} W_{jk} Q_k + \frac{\partial \dot{c}_i}{\partial q_j} \dot{q}_j + \frac{d^2 c_i}{dt^2} \qquad [3]$$

which is a matrix representation of a system of linear equations with the force vector C unknown. Noting that $\frac{\partial c_i}{\partial q_j}$ is an $n \times m$ matrix and W is an $m \times m$ matrix (where n is the number of constraint equations and m is the number of independent variables), we can see that equation [3] represents n equations and m unknowns. Since in general $n < m$, we have fewer equations than unknowns. We still need more information.

The fact that there are fewer equations than unknowns indicates that the system is underconstrained, which is as it should be. If the system were completely constrained, nothing could move. In an underconstrained system such as this, there exist many values for the constraint force C which would satisfy the equation. However, we want to add only enough force to cancel out any component of Q which would cause the object to deviate from the constraints.

To satisfy the constraints, the object can move only along the tangent planes to the surfaces $c_i = 0$. Thus, a constraint force that lies along the gradient to the constraints c will cancel any illegal force while not adding or deleting energy from the system. This observation yields:

$$C_j = \lambda_i \frac{\partial c_i}{\partial qj}, \qquad [4]$$

where λ is vector of scalars. The components of λ are known as *Lagrange multipliers*, and this technique of inverse dynamics is sometimes referred to as *the Lagrange multiplier method.* (See [Witkin, Gleischer, and Welch 1989] for a discussion of *the principle of virtual work.*)

We now have:

$$-[\frac{\partial c_i}{\partial q_j} W_{jk} \frac{\partial c_r}{\partial q_k}]\,\lambda_r = \frac{\partial c_i}{\partial q_j} W_{jk}\, Q_k + \frac{\partial \dot{c}_i}{\partial q_j}\,\dot{q}_j + \frac{d^2 c_i}{dt^2} \qquad [5]$$

Note $\frac{\partial c_i}{\partial q_j}$ is the Jacobian matrix for the constraint equations. (That is, each row i in the matrix represents that gradient vector for constraint c_i.) W_{jk} is the inverse of the mass matrix, and $\frac{\partial c_r}{\partial q_k}$ is the transpose of the Jacobian matrix. This gives us, on the left-hand side of equation [5], an $n \times m$ matrix multiplied by an $m \times m$ multiplied by an $m \times n$, yielding an $n \times n$ matrix. λ_r is an $n \times 1$ vector of unknowns. On the right-hand side, we get an $n \times 1$ vector of known values. We can now solve this system of linear equations for λ_r. Once λ_r is known, we can get C from equation [4], and we can finally solve for $\ddot{q}$.

Theoretically, the above solution should supply a constraint force sufficient to ensure that the objects always maintain their constraints. However, due to errors introduced in discretizing the integration, it becomes necessary to include a "feedback" term which inhibits drift. Thus the total force becomes:

$$Q_j + C_j + \alpha c_i \frac{\partial c_i}{\partial q_j} + \beta\, c_i \frac{\partial \dot{c}_i}{\partial q_j},$$

where α and β are constants.

Application of this method can be illustrated with a simple example. We will create a "midpointline" like the one constructed in

ThingLab (Section 3.3.2.2), with another fixed-length line attached at an endpoint. We will apply an initial force to the unattached endpoint of the midpointline and watch it move in response to the force.

Let $q = [q_1\ q_2\ q_3\ q_4\ q_5\ q_6\ q_7\ q_8]$ be the vector of independent variables with initial values [0 0 10 0 20 0 20 10]. That is, we have four points (q_1,q_2), (q_3,q_4), (q_5,q_6), (q_7,q_8) at initial positions (0,0), (10,0), (20,0), and (20,10), respectively. The first two constraints make the first three points collinear and equidistant. The third constraint fixes the length of the line between (q_5,q_6) and (q_7,q_8) at 10 units.

$$c_1:\ 2q_3 - q_1 - q_5 = 0$$
$$c_2:\ 2q_4 - q_2 - q_6 = 0$$
$$c3:\ 100 - (q_5-q_7)^2 - (q_6 - q_8)^2 = 0$$

The points are particles of unit mass. Thus the mass matrix is the identity matrix and can be dropped out of the equation. We will exert an initial force of 10 units in both the x and y directions on point $(q1,q2)$, which gives an initial force Q = [10 10 0 0 0 0 0 0].

The constraint Jacobian matrix is

$$\begin{bmatrix} -1 & 0 & 2 & 0 & -1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 2 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & -2(q_5-q_7) & -2(q_6-q_8) & 2(q_5-q_7) & 2(q_6-q_8) \end{bmatrix}$$

At the initial moment t_0, the second term on the right-hand side of equation [5], $\frac{\partial \dot{c}_i}{\partial q_j}\dot{q}_j$, will be 0 since the initial velocity is 0. For time $t > t_0$, we have $\frac{\partial \dot{c}_i}{\partial q_j}\dot{q}_j = \frac{\partial}{\partial q_j}(\frac{d}{dt}(c_i(q_r(t))))\ \dot{q}_j = \frac{\partial}{\partial q_j}(\frac{\partial c_i}{\partial q_r}\dot{q}_r)\ \dot{q}_j$.

The third term on the right-hand side, $\frac{d^2 c_i}{dt^2}$, will always be 0 in this example since the constraint functions do not depend on time.

Using a time increment of 0.1, we see the following motion of the object. (Time $t = 1$ occurs after 10 time slices.)

t	q 1	q 2	q 3	q 4	q 5	q 6	q 7	q 8
0	0.00	0.00	10.00	0.00	20.00	0.00	20.00	10.00
1	83.00	82.00	10.33	36.00	19.83	-0.09	20.00	9.91
2	1.67	1.64	10.67	73.00	19.67	-0.18	20.00	9.82
3	2.50	2.45	11.00	1.09	19.50	-0.27	20.00	9.72
4	3.33	3.27	11.33	1.46	19.33	-0.35	20.00	9.63
5	4.17	4.09	11.67	1.82	19.17	-0.44	20.00	9.53
6	5.00	4.91	12.00	2.19	19.00	-0.52	20.00	9.42
7	5.83	5.72	12.33	2.55	18.83	-0.60	20.00	9.33
8	6.67	6.54	12.67	2.92	18.67	-0.69	20.00	9.23
9	7.50	7.36	13.00	3.29	18.50	-0.77	20.00	9.12
10	8.33	8.17	13.33	3.66	18.34	-0.84	20.00	9.02

Table 2. Positions of midpointline in Lagangian method

Figure 22 shows the initial and final positions of this midpoint example. The force on the first point has caused it to move in a 45° angle, upward and to the right. This is consistent with the fact that the force was equal and positive in the x and y directions. This force, and the constraint that the second line maintain a fixed length, has caused the midpoint line to compress from a length of 20 to a length that is slightly less than 14, whereas the fixed length line retains its length of 10.

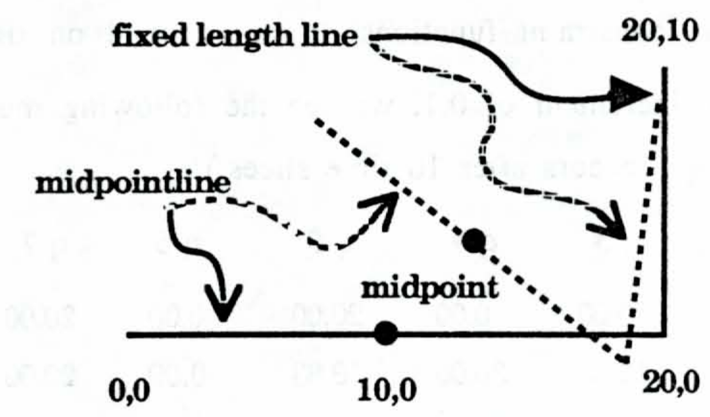


Figure 22. Lagrangian approach to midpoint with attached fixed length line. Solid lines are at t=0, dotted are at t=10.

3.3.4.3. More Physically-Based Constraints

Isaacs and Cohen [1987] combine behavior functions, kinematic constraints, and inverse dynamics to control the motion of jointed figures. Behavior functions determine higher level goals of motion, like a hand reaching for an object or a car stopping to avoid a cliff. Kinematic constraints specify exactly where a part of an object is to go. Inverse dynamics techniques then determine the forces which would result in the goal of motion.

Barzel and Barr [1988] also use inverse dynamics on rigid bodies. They divide the modeling problem into two parts: moving the parts of an object so that the object satisfies an initially unmet constraint (causing the object to "self-assemble"); and maintaining the constraint as the object moves and interacts with other objects. A catalog of useful constraints and an explicit algorithm for computing the constraint forces are given.

Platt and Barr [1988] apply reaction constraints and augmented Lagrangian constraints to flexible models, that is, putty-like objects. Reaction constraints are used to model the collision of a flexible model with a polygonal model. Augmented Lagrangian constraints combine the penalty method and the Lagrange multiplier method.

Platt [1989] applies the constraint methods to neural networks. The neural networks are actually differential equations that can be solved using standard techniques from optimization theory and numerical analysis, or implemented directly as circuits. Platt also reviews the constraint methods applied to physically-based modeling.

4. CONSTRAINT LOGIC PROGRAMMING

We now turn to programming languages which bring together finite- and continuous-domain constraint satisfaction. These languages are similar in that they are Prolog-based, and they attempt to establish a consistent framework for constraints defined over a variety of domains.

We saw in Section 2.3 that a Prolog program can express a consistent labeling problem, i.e., a finite-domain CSP. However, while the domains of the variables in the puzzle problem are implicit in the program, it is not possible to declare a finite domain in Prolog. Thus, domain-pruning strategies which help to reduce the search are not directly applicable in Prolog.

Furthermore, because Prolog is based upon syntactic matching and unification, it has only limited utility in expressing equalities and inequalities as constraints. Equalities can be handled as assignment statements, and inequalities as tests which either succeed or fail, but statements such as $X = Y$ or $X < 3$ fail in Prolog if X and Y are unbound.

Consider, for example, the following program, which computes the "triangular" number $R = \sum_{i=1}^{n} i = \frac{N(N+1)}{2}$.

```
triangular(0,0).
triangular(N,R) :-
    N1 = N - 1,
    triangular(N1,R1),
    R = N + R1.
```

This program illustrates Prolog's inability to express the bi-directionality we expect in a constraint problem. Given the query

?- triangular(4,R),

the program responds with $R = 10$. However, the program cannot find a solution to the query

?- triangular(N,10).

This is because $N1 = N-1$ is handled as an assignment statement rather than a constraint, and since N is unbound, no assignment is possible.

Recognition of Prolog's limitations as a language for stating constraint problems has led to the evolution of *constraint logic programming languages* [Cohen 1990]. What we would like in the program above is the ability to treat equalities and inequalities as true constraints. The following program computes the triangular number, but in the style of a constraint logic programming language:

```
triangular(0,0).
triangular(N, N+R) :-
    N ≥ 1,
    triangular (N-1, R).
```

We can now ask for a solution to the query

?- triangular(N, 10).

Like Prolog, a constraint logic programming language proceeds by matching and unification, but there is an additional mechanism for collecting constraints and checking their satisfiability each time a match is found. We can see how this operates in a short trace of this program.

The initial query cannot be matched to the first clause, *triangular(0,0)*, because the constant 10 in the query does not match the constant 0 in the clause. Thus the second clause is matched. As the match is made, an equality constraint is created between 10 and $N+R$. Also, the constraint $N \geq 1$ is added to the set of constraints. Within this clause, a recursive call, *triangular(N-1,R)*, is made. This call could not be handled in standard Prolog because N is unbound. However, in a constraint logic programming language, an equality constraint is simply created between $N-1$ and the variable to which it is eventually matched. (Variables are renamed in recursive matchings.) The constraint set is checked for satisfiability each time a matching takes place. For the next two calls to *triangular*, *triangular(0,0)* fails to match because of the constraints created upon matching. When the constraints can finally be satisfied by the match *triangular(0,0)*, values propagate through the constraints until the solution $N = 4$ is found.

Constraint logic programming languages allow for a consistent handling of constraints defined over a variety of domains. Three recently developed constraint logic programming languages are discussed in the following sections.

4.1. Prolog III

Prolog III, developed by the Groupe d'Intelligence Artificielle in Marseille, is a constraint logic programming language which allows constraints to be defined over rational numbers, booleans, and lists [Colmerauer 1990]. Each domain has associated with it appropriate operations and relations. Operations include +, -, *, $\wedge$, $\vee$, $\neg$, and $\cdot$ (concatenation). Linear equalities, inequalities, and disequalities can be defined over rational terms; equalities, disequalities, and subset relations can be defined over Boolean terms; and equalities and disequalities can be defined over lists. The most recent version of Prolog III also has facilities for calculating maximum and minimum values of numerical expressions.

We saw in Section 2.3 that the original Prolog implementations were based upon Robinson's unification algorithm. Prolog III integrates constraint satisfaction into the unification algorithm. This entails checking constraints for satisfiability as matching and unification proceed.

Colmerauer describes the execution of a Prolog III program in terms of an abstract machine. Essentially, a logic program can be viewed as a sequence of context-free rewriting rules. The initial state of the machine is represented by *(W, t_0 t_1 ... t_n, S)*, where *W* is a set of variables whose values are to be determined, t_0 t_1 ... t_n is a sequence of terms which we are trying to erase in the course of rewriting, and S is a system of constraints to be satisfied. (Terms correspond to the literals of the Prolog program, e.g. *triangular(N,R)*. Constraints are defined using the relevant operators and relations of the variable domains, e.g. $z = x + 2 * y$, or $a = \neg b \vee c$.)

A rule which is used to change the state of the machine is of the form

$s_0 \rightarrow s_1 \ldots s_m, R,$

where $s_0 \ldots s_m$ are terms, and R is a set of constraints.

Application of a rule yields a new state:

$(W, s_1 \ldots s_m\, t_1 \ldots t_n, S \cup R \cup \{t_0 = s_0\})$.

That is, the terms $s_1 \ldots s_m$ replace t_0 in the list of terms to be rewritten, the constraints associated with the rule are added to the set of constraints, and equality constraints are created or variables are given values as a result of matching a rule to a term. Importantly, the transition to the new state is permitted only if the constraints are satisfiable given the values assigned to variables thus far. Rewriting continues until all terms are eventually rewritten as unit clauses, which in turn can be erased. When the machine reaches a state in which all terms have been erased, it produces an answer on the basis of the remaining constraints.

To illustrate Prolog III's handling of constraints over rational numbers, we will solve an electrical circuit problem similar to the one solved above in Bertrand (Section 3.3.1). We can now add to the problem description some choices of available resistors and batteries, and we can limit the voltage drop across the resistors. (This program actually combines features of the Bertrand program and an electrical circuit program presented in [Jaffar and Michaylov 1987].) The program is shown in Listing 5.

```
availResistor(10). availResistor(14). availResistor(27).
availResistor(60). availResistor(100).

availCell (10). availCell(20).

battery (Volt, Vin, Vout) :-
      availCell (Volt),
      -Volt = Vin - Vout,
      Vin = 0.

resistor (Vin, Vout, Current, Resistor) :-
      availResistor(Resistor),
      Vin = (Current * Resistor) + Vout.

circuit (V, R1, R2, C) :-
      battery (V, V1, V2),
      resistor (V2, V3, C, R1),
      resistor (V3, V1, C, R2),
      V3-V1 < 17,
      V3-V1 > 14.
```

Listing 5. Prolog III code for circuits problem

The Prolog III program can offer multiple solutions to this problem based upon the available resistors and batteries. The solutions are $V=20$, $R1 = 10$, $R2 = 27$, and $C = \frac{20}{37}$; $V = 20$, $R1 = 14$, $R2 = 60$, and $C = \frac{20}{74}$; or $V = 20$, $R1 = 27$, $R2 = 100$, and $C = \frac{20}{127}$.

Because of the complexity of algorithms for solving constraints on integers, the structure underlying Prolog III does not contain a relation restricting a number to be an integer. However, there is a system-supplied predicate, *enum(X)* which effectively enumerates integers that might satisfy a set of constraints. Thus, a query which determines how to get 45 cents with no more than 8 American coins is as follows:

?- enum(q), enum(d), enum(n), enum(p),

$\{q + d + n + p \leq 8, 25q + 10d + 5n + p = 45, q \geq 0, d \geq 0, n \geq 0, p \geq 0\}.$

Prolog III also allows constraints to be defined over lists and Boolean values.

An important decision in the implementation of any constraint logic programming language is the choice of satisfiability algorithms to be used for the various domains. Cost is a consideration, since these algorithms are usually computationally expensive. Since constraints are added incrementally to the constraint set, incremental algorithms are generally sought. Simplification of the constraint set and the use of canonical forms for constraints can cut down on computation. Simplified forms are also desirable when solutions are presented in symbolic form.

The Prolog III numerical module uses an incremental simplex algorithm which can operate on linear constraints. This algorithm has the advantage of being able to detect the variables which have only one possible solution as it attempts to determine if the constraints are satisfiable.

The Boolean algebra module converts propositions into clausal form and uses SL-resolution [Kowalski and Kuehner 1971]. The conversion to clausal form is costly, but it has the advantage that, if the constraints are satisfiable, a simplified set of constraints containing only a minimal subset of variables can be output.

A prototype of the Prolog III interpreter has been implemented by Colmerauer and his colleagues at Université Aix-Marseille II, and a commercial version is now being distributed.

4.2. CLP($\mathfrak{R}$)

While Prolog III incorporates constraints over different domains into one constraint logic programming language, CLP($\mathfrak{D}$) is a family of constraint logic programming languages classified according to the domain over which constraints are defined. The parenthetical $\mathfrak{D}$ in CLP($\mathfrak{D}$) is a parameter denoting the type of constraints which can be handled by the language. Each domain has operations naturally associated with it, such as set union, logical disjunction, or multiplication, and constraints are expressed in terms of these operators [Jaffar and Lassez 1987].

In CLP($\mathfrak{R}$), for example, equality and inequality constraints are expressed in the domain of real numbers with the standard arithmetic operators (+,*, -,/) [Heintze et al. 1987]. Prolog is CLP($\mathfrak{H}$), where $\mathfrak{H}$ is the domain of finite trees subject to equality constraints. The portion of Prolog III which handles rational numbers can be viewed as CLP(Q). Another recent Prolog extension is referred to as CLP(Conceptual Theory) [Beringer and Porcher 1989]. CLP(Σ*) is constraint logic programming with regular sets [Walinsky 1989]. In this section, we will discuss CLP($\mathfrak{R}$) as a representative of the CLP($\mathfrak{D}$) family of languages.

Prolog III and CLP($\mathfrak{R}$) are very similar in syntax and implementation. Like Prolog III, CLP($\mathfrak{R}$) replaces syntactic unification with a more general constraint satisfaction mechanism. Rules in CLP($\mathfrak{R}$) are similar to Prolog clauses except that they can contain arithmetic expressions as well as atoms in the body. Both CLP($\mathfrak{R}$) and Prolog III are more powerful than Prolog in that they can handle equalities and inequalities among unbound variables, and they can give solutions in an implicit form, e.g., $x > 2$.

The CLP($\mathfrak{R}$) interpreter consists of a Prolog-like inference engine, a constraint solver, and an interface module. The inference engine operates in the manner described above for the Prolog III program: A CLP($\mathfrak{R}$) program begins with a goal and an initially-empty set of constraints. When a term is matched to the head of a clause, an equality constraint is created between unified variables or unified variables and constants. Any constraints encountered within the body of a matched clause are added to the set of constraints. However, unlike Prolog III, which adds all the constraints to the constraint set as soon as the clause is matched, CLP($\mathfrak{R}$) adds constraints only as they are encountered.

The interface simplifies the set of constraints, transforming them to a canonical form. In doing so, the interface may be able to determine the solvability of the constraints. If it cannot, the interface sends the constraints to the solver, which determines the set's solvability. The solver can handle linear and slightly non-linear equations (where evaluation can be delayed until enough information is available to make the set linear). If the set is not solvable, the search backtracks in the usual Prolog manner. If the set is solvable, a solution is produced.

Where Prolog III handles constraints over rational terms, CLP($\Re$) can deal with real numbers. As an example of CLP($\Re$)'s handling of numerical constraints, consider the following problem. A bridge constructed of piers and spans is to be built across a lake. The bridge must be at least 200 meters in length. Each pier costs \$100. Spans can be purchased in lengths that are multiples of 5 meters according to the price list in Table 3.

span length	**cost**
5 m	\$70
10 m	\$160
15 m	\$260
20 m	\$360
25 m	\$470
30 m	\$590
35 m	\$710
40 m	\$830
45 m	\$960
50 m	\$1090

Table 3. Cost of bridge spans

The problem is to determine the cost of the bridge given the length of spans to be used. The CLP($\Re$) program is presented in Listing 6.

```
bridge (Spanlength, Cost) :-
        span (Spanlength, Spancost),
        Piercost = 100,
        Length = 200,
        Numspans * Spanlength >= Length,
        (Numspans - 1) * Spanlength < Length,
        findit (Numspans),
        Cost = Numspans * Spancost + (Numspans + 1) * Piercost.

findit(X) :-
        X > 1, X <= 40,
        findit(X-1).
findit(1).

span (5, 70).
span (10, 160).
span (15, 260).
span (20, 360).
span (25, 470).
span (30, 590).
span (35, 710).
span (40, 830).
span (45, 960).
span (50, 1090).
```

Listing 6. CLP($\Re$) code for bridge

There are multiple solutions to this problem depending on which span type is chosen. The CLP($\Re$) system always answers with a *yes*, a *no*, or a *maybe*, denoting that the constraints are satisfiable, unsatisiable, or indeterminate, respectively. A *maybe* simply means that the system cannot determine if the constraints are satisfiable. If the answer is *yes*, a solution is produced. The solution may show values for all variables, or, if this is not possible, it may show a simplified version of the final set of constraints. Each time a solution is produced, the system asks the user if another solution is desired.

Prolog III would handle this problem in a similar manner, returning the solution in rational terms. However, there is one notable difference. In Prolog III, all constraints are collected and added to the constraint set as soon as a clause is matched. Thus constraints are written at the end of the clause, as in

```
findit(X) :-
    findit(X-1),
    X > 1, X <= 40.
```

If the *findit(X)* clause were written in this manner in CLP($\Re$), the program would never terminate because, not having the constraint as a condition under which the clause can match, the recursive calls would go on infinitely.

Linear equalities are handled in CLP($\Re$) by Gaussian elimination. Like Prolog III, CLP($\Re$) handles linear inequalities with an adaptation of the simplex algorithm. The implementation is made more efficient by a delay mechanism, which postpones the test for satisfiability of non-linear equalities and inequalities in the expectation that they will later become linear.

4.3. CHIP

CHIP (Constraint Handling in Prolog) is yet another constraint logic programming language which has evolved from Prolog [Van Hentenryck 1989a]. Like CLP($\Re$) and Prolog III, CHIP begins with a Prolog-style syntax and implementation, but it extends logic programming to new computation domains: specifically, finite domains, Boolean terms, and rational terms. CHIP also uses a type of depth-first branch-and-bound technique to solve discrete optimization problems.

CHIP provides efficient constraint-solving techniques applicable to each domain. Equality constraints over Boolean terms are handled by a unitary unification algorithm. Linear equalities, inequalities, and disequalities over rational terms are solved with a symbolic simplex-like algorithm. Details of these implementations and a discussion of their applications can be found in [Simonis, Nguyen, and Dincbas 1988] and [Graf 1987], respectively.

A large portion of the work on CHIP has been devoted to extensions of logic programming which make it more amenable to finite-domain constraint satisfaction. A central idea in the design of CHIP is the ability to declare finite domains for variables. We saw in Section 2.3 how a simple constraint satisfaction problem can be formulated as a Prolog program. In this example, the domains of the puzzle positions arise implicitly from the definition of the *neighbors* predicate. In CHIP, the domains for the variables in a predicate p can be declared explicitly with:

$$domain\ p(d_1,\ldots,d_n),$$

where d_i is the domain of the *i-th* variable. Each domain d_i can be either a constant, a set of constants or strings, or a sequence of natural numbers. If the *i-th* parameter of p is a list of variables, d_i applies to all the variables in the list.

The declaration of domains in CHIP makes possible a kind of *a priori* reduction of the search space in finite-domain constraint satisfaction through the application of *consistency checks*. CHIP offers *forward checking*, *looking ahead*, and *partial looking ahead* (Section 2.5.8) as a means of pruning the domains of the constrained variables, eliminating values which cannot contribute to a solution.

In general, application of the consistency checks in CHIP is left within the control of the programmer, who can write the logic program such that the search for a solution is performed more efficiently. Additionally, specializations of the consistency techniques provide for efficient handling of commonly used constraints. Notably, disequalities (e.g. $X \neq Y$) are handled by forward checking; linear equalities and inequalities (e.g. $3X + 2Y = 5Z$) are handled by partial looking ahead; and the constraint *element(I,L,X)*, which holds if the *I-th* element of the list *L* is *X* (where *L* is a list of integers and *I* and *X* are integers), is handled by looking ahead. Details of these techniques will be discussed below.

We begin with an example to illustrate how forward checking is applied. The problem is to complete the crossword puzzle in Figure 23. A variable is associated with each word place. Each variable has a domain of words which are the correct size to fill the given space. (In this case, all the words happen to be the same size.) Constraints of two types are placed on the words: Each word can be used only once, and intersecting words must have the same letter at the positions of intersection.

While the structure of the program is not dictated, CHIP is used most effectively in a kind of *generate and test* style. That is, the constraints between variables are generated first. Then values for the variables are generated and tested against the constraints. The program in Listing 7 follows this generate and test format.

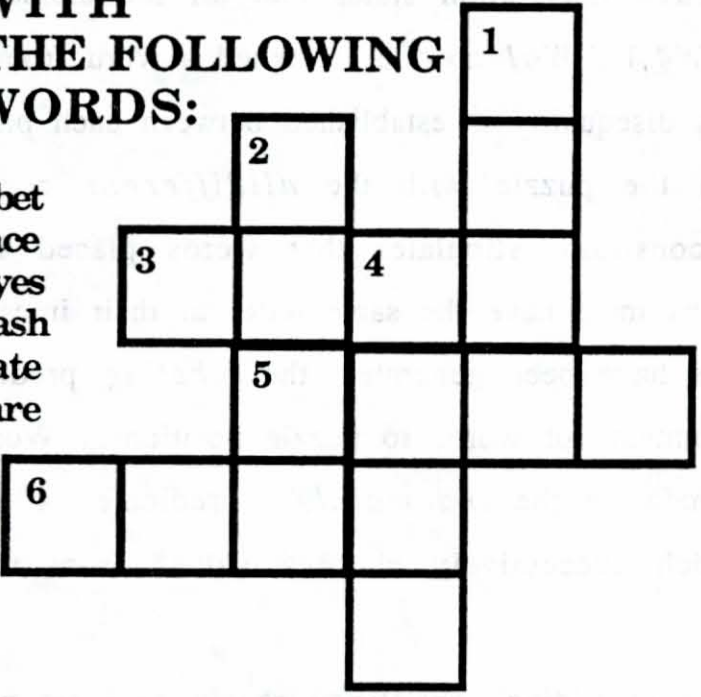
The *domain* declaration states that all the elements in the list *[W1,W2,W3,W4,W5,W6]* must be assigned a word from the given list of strings. A disequality is established between each pair of words to be placed in the puzzle with the *alldifferent* constraint. The *sameletter* constraint stipulates that words placed in intersecting puzzle positions must have the same letter at their intersection. Once the constraints have been generated, the *labeling* predicate generates possible assignments of words to puzzle positions. Words are chosen from the domain by the *indomain(X)* predicate, a system-defined predicate which successively chooses values from the domain of variable *X*.

One effect of adding consistency checks to logic programming is that a predicate can establish a constraint while not yet making the full commitment of binding values to variables. In effect, a constraint can be "deferred" until additional information makes it useful, at which time its implications are propagated through forward checking, looking ahead, or partial looking ahead. We can see how this is accomplished in the disequalities and *sameletter* constraints of the example above.

CROSSWORD PUZZLE:

FILL THE PUZZLE WITH THE FOLLOWING WORDS:

abet
lace
eyes
cash
rate
fare



SOLUTION:

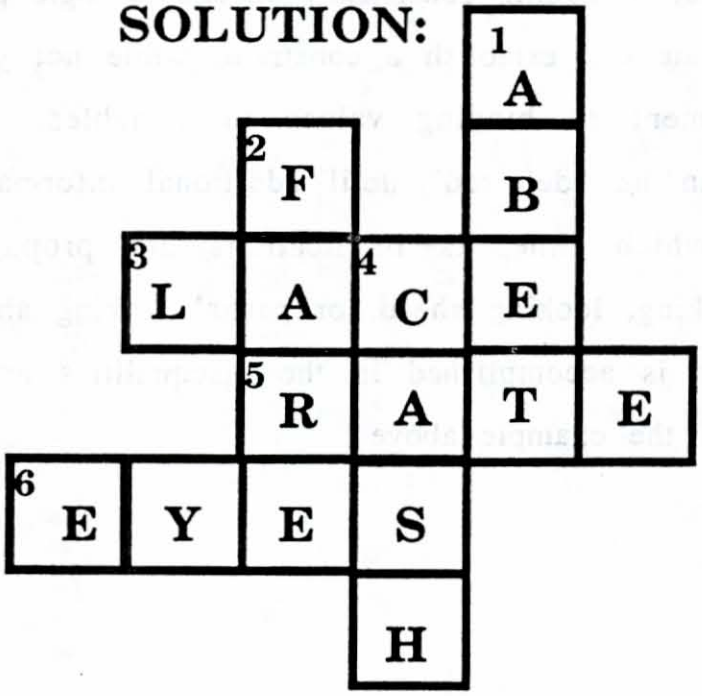


Figure 23. Constraints describing a crossword puzzle

```
alldifferent([]).
    alldifferent([X | Y]) :-
        outof(X,Y),
        alldifferent(Y).
    outof(X,[]).
    outof(X, [F | T]) :-
        X ≠ F,
        outof(X,T).
    forward  sameletter(g,d,g,d).
    sameletter(I,W1,J,W2) :-
        findletter(I,W1,Letter),
        findletter(J,W2,Letter).
    /*findletter finds the ith position in the string W2 and places it
     in Letter.  It can be handled by a Prolog system-defined
     predicate for string manipulation.*/
    labeling([]).
    labeling([X | Y]) :-
        indomain(X),
        labeling(Y).
    domain crossword ("abet", "lace", "eyes", "cash" "rate", "fare").
    crossword([W1,W2,W3,W4,W5,W6]) :-
        alldifferent ([W1,W2,W3,W4,W5,W6]),
        sameletter(3,W1,4,W3),
        sameletter(4,W1,3,W5),
        sameletter(2,W2,2,W3),
        sameletter(3,W2,1,W5),
        sameletter(4,W2,3,W6),
        sameletter(3,W3,1,W4),
        sameletter(2,W4,2,W5),
        sameletter(3,W4,4,W6),
        labeling([W1,W2,W3,W4,W5,W6]).
```

The program is initiated with the query:

?-crossword([W1,W2,W3,W4,W5,W6]).

Listing 7. CHIP solution to crossword puzzle

In standard Prolog, disequalities such as *W1* ≠ *W2* can be handled only when both variables are ground. Once both *W1* and *W2* are known, the constraint is simply used as a test which succeeds or fails. Disequalities in CHIP, on the other hand, are a specialization of the forward checking mechanism.

When a predicate is subject to forward checking, it is deferred until there remains only one variable in it which is not ground. At that time, the inference procedure makes use of the constraint by propagating the implications of the most recent binding through the constraint. The domain of the unbound variable is pruned to include only those values consistent with the values already assigned to the other variables.

Given the constraint *W1* ≠ *W2*, for example, if *W1* is bound to the value v, forward checking will prune *W2*'s domain *D* to *D-{v}*. If only one value remains in *W2*'s domain, *W2* is bound to that value. This is an important augmentation of forward checking, since the binding of a value can initiate additional pruning of domains. In the crossword puzzle example, forward checking is first used when the *indomain(X)* predicate binds *W1* to *abet*. The disequalities between *W1* and each other variable are then activated, and *abet* is removed from their domains.

Disequalities are automatically handled in CHIP by means of this forward checking mechanism, without any special declaration on the part of the programmer. It is also possible to specify that a user-defined predicate be handled in a forward-checking manner, as in the case of the *forward sameletter(g,d,g,d)* declaration. This declaration causes the *sameletter* constraint to be deferred until there remains only one domain-variable which is not ground, at which time the constraint is subject to forward checking. (The variables marked *d* in the *forward* declaration are domain-variables, i.e., variables with associated domains, while those marked *g* are expected to be ground by a constant.)

With forward checking, once *W1* is bound to *abet*, the constraint that the third letter of *W1* is equal to the fourth letter of *W3* will be forward checked, and only *lace*, *rate*, and *fare* will remain in the domain of *W3*. Similarly, *W5*'s domain will be pruned to include only *rate*, i.e., *W5* will be bound to *rate*. Because of the binding, all disequalities involving *W5* will again be forward checked, and *rate* will be removed from the domains of the other variables. Also, the binding of *W5* to *rate* initiates forward checking on the *sameletter(1,W5,3,W2)* and sameletter*(2,W5,2,W4)* constraints. Thus we see that with a forward-checking implementation, the search space is reduced *a priori*. In this example, no backtracking at all is required in the search.

Instead of the *forward* declaration on the *sameletter* constraint, a *lookahead* declaration could be used. A constraint $p(t_1,\ldots,t_n)$ is available to the lookahead mechanism if at least one of its terms t_i is a domain-variable (i.e., a domain has been declared for it), and each of the other terms is either ground or is a domain-variable. Unlike the forward checking mechanism, which waits for all but one variable to be ground, the looking ahead mechanism can prune domains as soon as the constraint is declared, and again whenever the binding of a variable provides new relevant information.

In CHIP, looking ahead considers in turn each *lookahead variable* v_i in a constraint and each element e_j in the domain of that variable. (A lookahead variable is a variable which is not ground, and which has an associated domain.) It retains e_j in the domain of v_i only if there is an assignment of values to the other variables in the constraint which is consistent with the assignment of e_j to v_i. Thus looking ahead can be viewed as enforcing k-consistency among the k lookahead variables in the constraint. (See Section 2.5.2.) Like forward checking, looking ahead binds a value to v_i if only one value remains in its domain.

In the crossword puzzle example, the *lookahead* declaration on the *sameletter* constraints will cause the domain of *W1* to be reduced immediately to *abet*, *eyes*, and *rate*; the domain of *W2* will be reduced to *lace*, and *fare*; the domain of *W3* will be reduced to *lace*; and so on. As in the forward-checking version, the problem is solved entirely by the consistency checks. On larger crossword puzzles, looking ahead generally is more successful at reducing the amount of backtracking, but at the expense of more consistency checks [Van Hentenryck 1989a].

The next example illustrates another domain-pruning technique applicable to linear equalities and inequalities. This technique is referred to as *partial lookahead.* We begin with some definitions.

A *linear equality (inequality)* is defined as an equality (inequality) between linear terms. A *linear term* is defined recursively as follows:

1. A natural number is a linear term.
2. A domain-variable ranging over natural numbers is a linear term.
3. $a*X + Y$ and $a*X - Y$ are linear terms if a is a natural number, X is a domain variable ranging over natural numbers, and Y is a linear term.

Consider now a linear inequality defined as a constraint. A linear inequality can be *normalized* into an expression of the form:

$$a_1X_1 + \ldots + a_nX_n + a = b_1Y_1 + \ldots + b_mY_m + b.$$

Let

$$a_1X_1 + \ldots + a_nX_n + a$$

range over $[min_1, max_1]$, and let

$$b_1Y_1 + \ldots + b_mY_m + b$$

range over $[min_2, max_2]$. Let min be the maximum of min_1 and min_2, and let max be the minimum of max_1 and max_2. Then to satisfy the constraint, both sides of the equation must range over $[min, max]$. From this we get:

$$a_1X_1 + \ldots + a_nX_n + a \geq min \text{ if } min_1 < min,$$

$$a_1X_1 + \ldots + a_nX_n + a \leq max \text{ if } max_1 > max,$$

and similarly for the right-hand side. We can now see that each variable X_i must satisfy:

$$a_iX_i \geq min - (\sum_{k \neq i} (a_k\, max_k) + a),$$

$$a_i X_i \leq max - (\sum_{k \neq i} (a_k\, min_k) + a),$$

where min_k and max_k are the minimum and maximum values respectively in the domain of X_k.

This same kind of reasoning can be applied to inequalities and results in significant *a priori* pruning of domains, as illustrated in the next example. The bridge problem solved above in CLP($\Re$) can now be solved in CHIP, with the additional feature of cost optimization. Given the query

bridge (Cost,Spanlength,Numspans, 200),

the following program will determine the minimum cost, the length of spans, and the number of spans needed to build a bridge 200 units in length.

As before, we set up the constraints before generating possible labelings. Domains and inequality constraints are established with *getNumspans*. The inequalities require that enough spans are used to cross the water, but only the minimum number needed to do so. Since no values in the domain of *Spanlength* will satisfy the first inequality constraint when *Numspans* is less than 4, the values {1,2,3} are deleted from the domain of *Numspans*. Similarly, since no values in the domain of *Spanlength* will satisfy the second inequality if *Numspans* is greater than 40, values greater than 40 are deleted from the domain of *Numspans*. More importantly, each time *labeling* generates values for *Spanlength* and *Spancost*, the domain of *Numspans* is restricted to just one value by the inequality constraints.

```
domain getNumspans (1..100, {5,10,15,20,25,30,35,40,45,50}, 200).
    getNumspans (Numspans, Spanlength, Length) :-
        Numspans * Spanlength >= Length,
        (Numspans - 1) * Spanlength < Length.

    labeling (5, 70).
    labeling (10, 160).
    labeling (15, 260).
    labeling (20, 360).
    labeling (25, 470).
    labeling (30, 590).
    labeling (35, 710).
    labeling (40, 830).
    labeling (45, 960).
    labeling (50, 1090).

    bridge (Cost,Spanlength,Numspans) :-
        Piercost = 100,
        getNumspans (Numspans, Spanlength, Length),
        Cost = Numspans * Spancost + (Numspans + 1) * Piercost,
        minimize (labeling(Spanlength,Spancost), Cost).
```

Listing 8. CHIP solution to bridge problem

The higher order constraint

minimize(labeling(Spanlength,Spancost),Cost)

results in the minimization of *cost* based upon the bindings which result from *labeling(Spanlength,Spancost)*. CHIP uses a simple depth-first branch and bound strategy that operates in the following manner: When a solution at cost *cost* is determined, a new constraint is introduced dynamically. This constraint requires that the cost of any other solution be better than (in this example, less than) *cost*. These new constraints are handled with the usual *a priori* pruning

SWAIN, M. J., AND COOPER, P. R. 1987. Parallel hardware for constraint satisfaction. *Proceedings AAAI* 87, *Vol. 1* (Seattle, Wash., July 13-17), pp. 682-686.

VAN HENTENRYCK, P. 1989a. *Constraint Satisfaction in Logic Programming.* MIT Press, Cambridge, Mass.

VAN HENTENRYCK, P. 1989b. Parallel constraint satisfaction in logic programming. *Proceedings of the 6th International Conference on Logic Programming* (Lisbon, Portugal, June 19-23), pp. 165-180.

VAN WYK, C. J. 1982. A high-level language for specifying pictures. *ACM Trans. on Graphics* 1,2 (April), 163-183.

WALINSKY, C. 1989. CLP(Σ*): constraint logic programming with regular sets. *6th International Conference on Logic Programming*, (Lisbon, Portugal, June 19-23), pp. 180-196.

WALTZ, D. 1975. Understanding line drawings of scenes with shadows. In *The Psychology of Computer Vision*, P. H. Winston, Ed. McGraw-Hill, New York.

WITKIN, A., FLEISCHER, K., AND BARR, A. 1987. Energy constraints on parameterized models. *Computer Graphics* 21, 4 (July), 225-229.

WITKIN, A., GLEICHER, M., AND WELCH, W. 1990. Interactive Dynamics. *Computer Graphics*, 24, 2 (March), 11-22.

WOLFRAM, S. 1988. *Mathematica: A System for Doing Mathematics by Computer*, Addison-Wesley, Reading, Mass.

techniques, resulting in active constraint propagation and a reduction of the search space.

CHIP has been used to solve a number of well-known finite-domain constraint satisfaction problems (e.g. N-queens, cryptarithmetic, graph coloring, and cutting stock problems), and Van Hentenryck shows through empirical results that solutions which benefit from the consistency techniques are significantly faster than standard logic programming solutions. A parallel implementation of CHIP (or, more precisely, of the portion of CHIP which uses consistency techniques for finite-domain constraint satisfaction) is now under way [Van Hentenryck 1989b]. This implementation exploits the or-parallelism inherent in constraint satisfaction, pursuing in parallel the multiple solutions to a problem. When applied to the optimization predicates, or-parallelism results in a parallel, depth-first branch-and-bound approach.

The CHIP project originated in 1985 at the European Computer-Industry Research Center (ECRC) in Munich, Germany, headed by Mehmet Dincbas. A Prolog-style interpreter has been implemented, and work has been done on compilation of CHIP. A prototype of the or-parallel/finite-domain portion of CHIP has been implemented on a Siemens MX500 (equivalent to a Sequent Balance 8000).

4.4. Concurrent Constraint Logic Programming

Yet another development in the evolution of logic programming is a class of *concurrent constraint logic programming languages*. This class of languages originated from two sources: constraint logic programming languages [Jaffar and Lassez 1987]; and concurrent logic programming languages such as Parlog [Clark and Gregory 1985] and Concurrent Prolog [Shapiro 1987].

Saraswat [1989] designs a class of concurrent constraint logic programming languages based upon the idea of *store as constraint*. That is, a constraint program defines the relationships among a set of variables ranging over some domain, and the *store* represents the constantly refined set of possible values for each variable at each step in the computation. Computation emerges from the interaction of concurrently executing agents which either place constraints on variables or ask if constraints are upheld by the current store. These actions, referred to respectively as *atomic Tell* and and *atomic Ask*, provide a channel of communication between executing agents which lends itself well to concurrency. Saraswat illustrates the usefulness of his approach by implementing standard algorithms such as mutual exclusion and many-server many-client communication. He also shows that his language can be used to solve a finite-domain CSP using fine-grained parallelism.

More recently, two specific languages have been proposed: Janus, a member of the family of distributed constraint programming languages (a subset of concurrent constraint programming languages), is designed to coordinate the actions of large numbers of diverse computations. (A discussion of Janus will appear in the Proceedings of the 1990 North American Conference on Logic Programming.) Lucy (a syntactic subset of Janus) is a very simple concurrent constraint language based upon the actor model of computation [Hewitt and Baker 1988]; [Clinger 1981]; [Agha 1985].

(A discussion of Lucy will appear in the 1990 OOPSLA Conference Proceedings.)

CONCLUSIONS

The wide-ranging applications of constraint-based problem formulations should come as no surprise. We see around us a world of complex relationships, but our vision is myopic. Those things which we can see clearly, we try to capture in a precise language, mathematical or logical. The idea is that this near-sighted view, when operated on by the rules which govern our "world," can lead to a more complete picture. In problem-solving, we get solutions which satisfy the relations among all the objects simultaneously. In modeling and simulation, we see realistic, though perhaps unpredicted, behavior emerge from object descriptions.

It should also come as no surprise that constraint problems are generally hard problems in terms of computational complexity. While it is often argued that a declarative rather than procedural formulation is a more "natural" way to express many problems, where we gain in "natural human expression" we generally lose in efficiency of computation. The complexity of constraint problems, as well as their relationship to human problem solving, point to parallel computation as the next exciting avenue for constraint-based programming. As research in this area continues, the challenge remains to make computers think the way we think, not only to solve problems, but to learn about problem-solving itself.

ACKNOWLEDGEMENTS

This work was supported in part by the Florida High Technology and Industry Council, and the U.S. Army's Program Manager for Training Devices. We thank Dr. Robert Brigham for his helpful editorial comments. We also thank Julie Carrington and Xin Li for their help in implementing the physical modeling examples.

REFERENCES

AGHA, G. 1985. Actors: a model of concurrent computation in distributed systems. PhD dissertation, Ann Arbor, Mi. University of Michigan.

BARZEL, R., AND BARR, A. 1988. A modeling system based on dynamic constraints. *Computer Graphics* 22, 4 (August), 179-187.

BERINGER, H., AND PORCHER, F. 1989. A relevant scheme for Prolog extensions: CLP(Conceptual Theory). *6th International Conference on Logic Programming* (Lisbon, Portugal, June 19-23), pp. 131-148.

BORNING, A. 1986. Defining constraints graphically. *Human Factors in Computing Systems CHI'86 Conference Proceedings* (April 13-17), pp. 137-143.

BORNING, A. 1981. The programming language aspects of ThingLab, a constraint-oriented simulation laboratory. *ACM Trans. Program. Lang. and Syst.* 3, 2 (Oct.), 353-387.

BORNING, A. 1979. ThingLab -- a constraint-oriented simulation laboratory. Ph.D. dissertation, Computer Science Dept., Stanford Univ., Stanford, Calif.

BORNING, ET AL. 1987. Constraint hierarchies. *OOPSLA '87 Proceedings* (Orlando, Fl., October 4-8), pp. 48-60.

BORNING, A., ET AL. 1989. Constraint hierarchies and logic programming. *6th International Conference on Logic Programming* (Lisbon, Portugal, June 19-23), pp. 149-164.

CHANG, CHIN-LIANG, AND LEE, RICHARD CHAR-TUNG. 1973. *Symbolic Logic and Mechanical Theorem-Proving*. Academic Press, New York.

CLARK, K. AND GREGORY, S. 1986. Parlog: parallel programming in logic. *ACM Trans. Program. Lang. and Syst.* 8, 1 (Jan.), 1-49.

CLINGER, W. L. 1981. Foundations of Actor semantics. PhD thesis, MIT, Cambridge, Mass.

CLOCKSIN, W.F., AND MELLISH, C.S. 1987. *Programming in Prolog*. 3rd ed. Springer-Verlag, New York.

COHEN, J. 1990. Constraint logic programming languages. *Commun. ACM* 33, 7 (July), 52-68.

COLMERAUER, A. 1990. An introduction to Prolog III. *Commun. ACM* 33, 7 (July), 69-90.

COOPER, P. 1988. Structure recognition by connectionist relaxation: formal analysis. *Proceedings of the 7th Biennial Conference of the Canadian Society for Computational Studies of Intelligence. CSCSI '88* (Edmonton, Alberta, Canada, June 6-8), pp. 148-155.

DAVIS, E. 1987. Constraint propagation with interval labels. *Artificial Intell.* 32, 3 (July), 281-331.

DAVIS, L. S., AND ROSENFELD, A. 1981. Cooperating processes for low-level vision: a survey. *Artificial Intell.* 17, 1-3 (August), 245-263.

DECHTER, R. 1988. Constraint processing incorporating backjumping, learning, and cutset-decomposition. *IEEE 4th Conf. on AI Applications* (San Diego, Calif., March 14-18), pp. 312-317.

DECHTER, R. 1986. Learning while searching in constraint-satisfaction-problems. *Proceedings AAAI 86, Vol. 1* (Philadelphia, Pa., Aug. 11-15), pp. 178-183.

DECHTER, R., AND DECHTER A. 1987. Belief maintenance in dynamic constraint networks. *Proceedings AAAI* 87, *Vol. 1* (Seattle, Wash., July 13-17), pp. 37-42.

DECHTER, R., AND PEARL, J. 1988. Network-based heuristics for constraint satisfaction problems. *Artificial Intellig.* 34, 1 (Dec.), 1-38.

DESCOTTE, Y., AND LATOMBE, J.-C. 1985. Making compromises among antagonist constraints in a planner. *Artificial Intellig.* 27, 2 (Nov.), 185-217.

DOYLE, J. 1979. A truth maintenance system. *Artificial Intellig.* 12, 3 (Nov.), 231-272.

DUISBERG, R. A. 1986. Constraint-based animation: temporal constraints in the Animus system. Tektronix Laboratories Technical Report No. CR-86-37, Aug.

EGE, R. K. 1989. Direct manipulation user interfaces based on constraints. *Proceedings of 13th Internat'l IEEE COMPSAC Conference* (Orlando, FL, Sept.), 374-380.

EGE, R. K., MAIER, D., AND BORNING, A. 1987. The filter browser: defining interfaces graphically. *Proceedings of the European Conference on Object-Oriented Programming* (Paris, France, June), 155-165.

FIKES, R. E., AND NILSSON, N. J. 1971. STRIPS: a new approach to the application of theorem proving to problem solving. *Artificial Intellig.* 2, 189-205.

FREEMAN-BENSON, B. N., MALONEY, J., AND BORNING, A. 1990. An incremental constraint solver. *Commun. ACM* 33, 1 (Jan.), 54-63.

FREUDER, E. C. 1978. Synthesizing constraint expressions. *Commun. ACM* 21, 11 (Nov.), 958-965.

GASHNIG, J. 1974. A constraint satisfaction method for inference making. *Proceedings of the 12th Allerton Conf. On Circuit and System Theory* (Urbana, Ill., Oct. 2-4), pp. 866-875.

GASHNIG, J. 1978. Experimental case studies of backtrack vs. Waltz-type vs. new algorithms for satisficing-assignment problems. *Proceedings of the 2nd National Conference of the Canadian Society for Computational Studies of Intelligence.* CSCSI '78 (Toronto, Ont., July 19-21), pp. 268-277.

GASHNIG, J. 1977. A general backtrack algorithm that eliminates most redundant tests. *Proceedings of the 5th International Joint Conference On Artificial Intelligence.* IJCAI (Cambridge, Mass., Aug. 22-25), p. 457.

GOLDBERG, A., AND ROBSON, D. 1983. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, Reading, Mass.

GOLDSTEIN, H. 1965. *Classical Mechanics.* Addison-Wesley, Reading, Mass.

GOTTFRIED, B. S., AND WEISMAN, J. 1973. *Introduction to Optimization Theory.* Prentice-Hall, Englewood Cliffs, N. J.

GRAF, T. 1987. Extending constraint handling in logic programming to rational arithmetic. European Computer-Industry Research Centre (ECRC) Internal Report, Munich, Germany, 1987.

GROSSMAN, M., AND EGE, R. K. 1987. Logical composition of object-oriented interfaces. *OOPSLA '87 Proceedings* (Orlando, Fl., October 4-8), pp. 295-304.

HARALICK, R., ET AL. 1978. Reduction operations for constraint satisfaction. *Inform. Sci.* 14, 199-219.

HARALICK, R. M., AND ELLIOTT, G. L. 1980. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intellig.* 14, 3 (Oct.), 263-313.

HARALICK, R. M., AND SHAPIRO, L. G. 1979. The consistent labeling problem: part I. *IEEE Trans. Pattern Anal. Machine Intell. PAMI-1*, 2 (April), 173-184.

HAYES-ROTH, B., ET AL. 1986. Protean: deriving protein structure from constraints. *Proceedings AAAI 86, Vol. 1* (Philadelphia, Pa., Aug. 11-15), pp. 904-909.

HEINTZE, N., ET AL. 1987. *The CLP(ℜ) Programmer's Manual*, Version 2.0. Dept. of Computer Science, Monash University, Clayton 3168, Victoria, Australia, June.

HESTENES, M. 1975. *Optimization Theory*. Wiley & Sons, New York.

HEWITT, C., AND BAKER, H. 1978. Actors and continuous functionals. In E. J. Neuhold, Ed. *Formal Descriptions of Programming Concepts*. North-Holland Publishing Co., New York, pp. 367-390.

HUMMEL, R. A., AND ZUCKER, S. W. 1983. On the foundations of relaxation labeling processes. *IEEE Trans. Pattern Anal. Machine Intell. PAMI-5*, 3 (May), 267-287.

ISAACS, P., AND COHEN, M. 1987. Controlling dynamic simulation with kinematic constraints, behavior functions, and inverse dynamics. *Computer Graphics* 21, 4 (July), 215-224.

JAFFAR, J. AND LASSEZ, J-L. 1987. Constraint logic programming. *Proceedings of 14th ACM Symposium on Principles of Programming Languages* (Munich, Germany, Jan.).

JAFFAR, J., AND MICHAYLOV, S. 1987. Methodology and implementation of a constraint logic programming system. *Proceedings of the 4th International Conference on Logic Programming* (Melbourne, Australia, May), pp. 196-219.

KASIF, SIMON. 1986. On the parallel complexity of some constraint satisfaction problems. *Proceedings AAAI 86, Vol. 1* (Philadelphia, Pa., Aug. 11-15), pp. 349-353.

KOWALSKI, R., AND KUEHNER, D. 1971. Resolution with selection function. *Artificial Intellig.* 3, 3, 227-260.

KNUTH, D. E. 1979. *TeX and METAFONT: New Directions for Typesetting*. American Mathematical Society and Digital Press.

LELER, W. 1988. *Constraint Programming Languages: Their Specification and Generation*. Addison-Wesley, Reading, Mass.

LLOYD, J.W. 1987. *Foundations of Logic Programming*. 2nd ed. Springer-Verlag, New York.

LOVELAND, D. 1978. *Automated Theorem Proving: A Logical Basis.* North-Holland Publishing Company, New York.

MACKWORTH, A. 1977. Consistency in networks of relations. *Artificial Intellig.* 8, 1 (Feb.), 99-118.

MACKWORTH, A. K., AND FREUDER, E. C. 1985. The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. *Artificial Intellig.* 25, 1 (Jan.), 65-74.

MALONEY, J. H., BORNING, A., AND FREEMAN-BENSON, B. N. 1989. Constraint technology for user-interface construction in ThingLab. *OOPSLA '89 Proceedings* (New Orleans, La., October 1-6), pp. 381-388.

MARON, M. J. 1982. *Numerical Analysis: A Practical Approach.* Macmillan, New York.

MINSKY, M. 1975. A framework for representing knowledge. In *The Psychology of Computer Vision*, P. H. Winston, Ed. McGraw-Hill, New York, pp. 211-277.

MOHR, R., AND HENDERSON, T. C. 1986. Arc and path consistency revisited. *Artificial Intellig.* 28, 2 (March), 225-233.

MONTANARI, U. 1974. Networks of constraints: fundamental properties and applications to picture processing. *Inform. Sci.* 7, 95-132.

MOSES, J. 1971. Algebraic simplification: a guide for the perplexed. *Commun. ACM.* 14,8 (Aug.), 527-537.

NELSON, G. 1985. JUNO, a constraint-based graphics system. *Computer Graphics*, (July), 235-243.

NEWELL, A., AND SIMON, H. 1972. *Human Problem Solving*. Prentice-Hall, Englewood Cliffs, N. J.

NUDEL, B. 1983. Consistent-labeling problems and their algorithms: expected complexities and theory-based heuristics. *Artificial Intellig.*, 21, 1-2 (March), 135-178.

O'DONNELL, M. J. 1985. *Equational Logic as a Programming Language*. MIT Press, Cambridge, Mass.

PLATT, J. 1989. Constraint methods for neural networks and computer graphics. PhD dissertation, California Institute of Technology.

PLATT, J., AND BARR, A. 1988. Constraint methods for flexible models. *Computer Graphics* 22, 4 (August), 279-288.

ROBINSON, J. A. 1965. A machine-oriented logic based on the resolution principle. *Journal of ACM* 12, 1 (Jan.), 23-41.

ROSENFELD, A., HUMMEL, R. A., AND ZUCKER, S. W. 1976. Scene labeling by relaxation operations. *IEEE Trans. Syst. Man Cybern. SMC-6*, (May), 420-433.

SARASWAT, V. A. 1989. Concurrent constraint programming languages. Ph.D. dissertation, Computer Science Dept., Carnegie Mellon Univ., Pittsburgh, Pa.

SHAMBLIN, J. E., AND STEVENS, G. T. 1974. *Operations Research: A Fundamental Approach*. McGraw-Hill, New York.

SHAPIRO, EHUD, ED. 1987. *Concurrent Prolog, Volume 1 and 2*. MIT Press, Cambridge, Mass.

SIMONIS, H., NGUYEN, H.N., AND DINCBAS, M. 1988. Verification of digital circuits using CHIP. *Proceedings of the IFIP WG 10.2 International Working Conference on the Fusion of Hardware Design and Verification*, Glasgow, Scotland, July.

STEFIK, M. 1981. Planning with constraints (MOLGEN: Part 1). *Artificial Intellig.* 16, 2 (May), 111-140.

SUSSMAN, G., AND STEELE, G. 1980. CONSTRAINTS--a language for expressing almost-hierarchical descriptions. *Artificial Intellig.* 14, 1 (Aug.), 1-39.

SUTHERLAND, I. 1963. Sketchpad: a man-machine graphical communication system. MIT Lincoln Laboratory Technical Report No. 296, Jan..

0000104