Program Does Not Equal Program: Constraint Programming and Its Relationship to Mathematical Programming

IRVIN J. LUSTIG
ilustig@ilog.com

ILOG
1080 Linda Vista Avenue
Mountain View, California 94043

JEAN-FRANÇOIS PUGET
puget@ilog.fr

ILOG
9 rue Verdun
BP 85
Gentilly Cedex, France

Arising from research in the computer science community, constraint programming is a fairly new technique for solving optimization problems. For those familiar with mathematical programming, a number of language barriers make it difficult to understand the concepts of constraint programming. In this short tutorial on constraint programming, we explain how it relates to familiar mathematical programming concepts and how constraint programming and mathematical programming technologies are complementary. We assume a minimal background in linear and integer programming.

George Dantzig [1963] invented the simplex method for linear programming in 1947 and first described it in a paper entitled "Programming in a linear structure" [Dantzig 1948, 1949]. Fifty years later, linear programming is now a strategic technique used by thousands of businesses trying to optimize their global operations. In the mid-1980s, researchers developed constraint programming as a computer science technique by combining developments in the artificial intelligence community with the development of new computer programming languages. Fifteen years later, constraint programming is now being seen as an important technique that complements traditional mathematical programming technologies as businesses continue to look for ways to optimize their business operations.

Developed independently as a technique within the computer science literature, constraint programming is now getting attention from the operations research com-

0092-2102/01/3106/0029/$05.00
1526–551X electronic ISSN
This paper was refereed.

PROGRAMMING—INTEGER—ALGORITHMS—BRANCH AND BOUND COMPUTERS/
COMPUTER SCIENCE—ARTIFICIAL INTELLIGENCE PROGRAMMING—CONSTRAINT

munity as a new and sometimes better way of solving certain kinds of optimization problems. We provide an introduction to constraint programming for those familiar with traditional mathematical programming.

The Word *Programming*

For those familiar with mathematical programming, one of the confusions with regard to understanding constraint programming is the fact that the names of both areas share the word *programming*. In fact, the two disciplines use this word differently.

The field of mathematical programming arose from its roots in linear programming. In his seminal textbook, Dantzig [1963, p. 1–2] introduces linear programming by describing a few different planning problems:

> Nevertheless, it is possible to abstract the underlying essential similarities in the management of these seemingly disparate systems. To do this entails a look at the structure and state of the system, and at the objective to be fulfilled, in order to construct a statement of the actions to be performed, their timing, and their quantity (called a "program" or "schedule"), which will permit the system to move from a given status toward the defined objective.

The problems that Dantzig studied while developing the simplex algorithm were "programming problems," because the United States Defense Department in the post-World War II era was supporting research to devise programs of activities for future conflicts. T. C. Koopmans suggested that Dantzig use the term *linear programming* as opposed to *programming in a linear structure*, and Al Tucker then suggested that Dantzig call the linear programming problem a *linear program* [Dantzig and Thapa 1997]. Therefore, the term *program*, previously used to describe a plan of activities, became associated with a specific mathematical problem in the operations research literature.

Constraint programming is often called constraint logic programming, and it originates in the artificial intelligence literature in the computer science community. Here, the word *programming* refers to computer programming. Knuth [1968, p. 5] defines a computer program as "an expression of a computational method in a computer language." A computer program can be viewed as a plan of action for the operations of a computer, and hence the common concept of a *plan* is shared with the planning problems studied in the development of the simplex method. Constraint programming is a computer programming technique, with a name that is in the spirit of other programming techniques, such as object-oriented programming, functional programming, and structured programming. Logic programming is a declarative, relational style of programming based on first-order logic, where simple resolution algorithms are used to resolve the logical statements of a problem. Constraint logic programming extends this concept by using more powerful algorithms to resolve these statements. Van Hentenryck [1999, p. 4] writes:

> The essence of constraint programming is a two-level architecture integrating a constraint and a programming component. The constraint component provides the basic operations of the architecture and consists of a system reasoning about fundamental properties of constraint systems such as satisfiability and entailment. The constraint component is often called the constraint store, by analogy to the memory store of traditional programming languages. . . . Operating around the constraint store is a programming-language component that speci-

fies how to combine the basic operations, often in non-deterministic ways.

Hence, a *constraint program* is not a statement of a problem as in mathematical programming, but is rather a computer program that indicates a method for solving a particular problem. It is important to emphasize the two-level architecture of a constraint programming system. Because it is first and foremost a computer programming system, the system contains representations of programming variables, which are representations of memory cells in a computer that can be manipulated within the system. The first level of the constraint programming architecture allows users to state constraints over these programming variables. The second level of this architecture allows users to write a computer program that indicates how the variables should be modified so as to find values of the variables that satisfy the constraints.

The roots of constraint programming can be traced back to the work on constraint satisfaction problems in the 1970s, with arc-consistency techniques [Mackworth 1977] on the one hand and the language ALICE [Lauriere 1978] that was designed for stating and solving combinatorial problems on the other hand. In the 1980s, work in the logic programming community showed that the Prolog language could be extended by replacing the fundamental logic programming algorithms with more powerful constraint solving algorithms. For instance, in 1980, Prolog II used a constraint solver to solve equations and disequations on terms. This idea was further generalized in the constraint logic programming scheme and implemented in several languages [Colmerauer 1990; Jaffar and Lassez 1987; Van Hentenryck 1989; and Wallace et al. 1997]. Van Hentenryck [1989] used the arc-consistency techniques developed in the constraint satisfaction problem framework as the algorithm for the basic constraint solving. This concept was termed *finite domain constraints*.

In the 1990s, researchers transformed constraint logic programming based on Prolog to constraint programming by providing constraint programming features in general-purpose programming languages. Examples includes Pecos for Lisp [Puget 1992], ILOG Solver for C++ [ILOG 1999], and Screamer for Common Lisp [Siskind and McAllester 1993]. This development made possible powerful constraint solving algorithms in the context of mainstream programming languages [Puget and Leconte 1995]. Another rich area of research in constraint programming has been the development of special-purpose programming languages to allow people to apply the techniques of constraint programming to different classes of problems. Examples include Oz [Smolka 1993] and CLAIRE [Caseau and Laburthe 1995]. In designing such languages, their developers have sought to provide complete languages for doing computer programming and hence the languages allow users to implement constraint solving algorithms. Departing from this approach, Van Hentenryck [1999] designed OPL (Optimization Programming Language) to make it easy to solve optimization problems by supporting constraint programming and mathematical programming techniques. He did not deem the completeness of the language for computer programming or the ability to program constraint solving algorithms as important. Instead, he designed the language to support the declarative representation of optimization problems, providing the facilities to use an underlying constraint programming engine combined with the ability to describe, via computer programming techniques, a search strategy for finding solutions to problems. The OPL language is not a complete computer programming language,

Constraint programming is a computer programming technique.

but rather a language that is designed to allow people to solve optimization problems using either constraint programming or mathematical programming techniques. An advantage of OPL is that the same language is used to unify the representations of decision variables from traditional mathematical programming with programming variables from traditional constraint programming. Some of the examples we present are related to those Van Hentenryck [1999] presents.

Van Hentenryck was motivated to design OPL by the increased interest in mathematical programming modeling languages, such as AMPL [Fourer, Gay, and Kernighan 1993] and GAMS [Bisschop and Meeraus 1982], and the recent use of constraint programming to solve combinatorial optimization problems. These NP-hard problems include feasibility problems as well as optimization problems. Constraint programming is often used when people want a quick feasible solution to a problem rather than a provably optimal solution. Because a constraint programming system provides a rich declarative language for stating problems combined with a programming language for describing a procedural strategy to find a solution, constraint programming is an alternative to integer programming for solving a variety of combinatorial optimization problems.

Van Hentenryck [1989], Marriott and Stuckey [1999], and Hooker [2000] describe some of the underlying theory of constraint programming. In his book about OPL, Van Hentenryck [1999] gives a number of examples of how constraint programming can be applied to real problems. Unfortunately, we have yet to find a good book that gives the techniques for applying constraint programming to solve optimization problems and is written in the spirit of the book by Williams [1999] for mathematical programming. Williams demonstrates a wide variety of modeling techniques for solving problems using mathematical programming.

Constraint Programming Formulations

To explain the constraint programming framework, we will first characterize the ways that constraint programming can be applied to solve combinatorial optimization problems by developing a notation to describe these problems. Then we will show formulations of feasibility problems and optimization problems using the OPL language.

We first formally define a constraint satisfaction problem, using some of the terminology of mathematical programming. Given a set of n decision variables x_1, x_2, . . . , x_n, the set D_j of allowable values for each decision variable x_j, $j = 1, \ldots, n$, is called the *domain* of the variable x_j. The domain of a decision variable can be any possible set, operating over any possible set of symbols. For example, the domain of a variable could be the even integers between 0 and 100 or the set of real numbers in the interval [1,100] or a set of people {Tom, Judy, Jim, Ann}. There is no restriction on the type of each decision variable, and hence decision variable can take on integer values, real values, set elements, or even subsets of sets.

Formally, a constraint $c(x_1, x_2, \ldots, x_n)$ is a mathematical relation, that is, a subset S of the set $D_1 \times D_2 \times \ldots \times D_n$, such that if $(x_1, x_2, \ldots, x_n) \in S$, then the constraint is said to be satisfied. Alternatively, we can define a constraint as a mathematical function $f: D_1 \times D_2 \times \ldots \times D_n \rightarrow \{0,1\}$ such that $f(x_1, x_2, \ldots, x_n) = 1$ if and only if $c(x_1, x_2, \ldots, x_n)$ is satisfied. Using this functional notation, we can then define a constraint satisfaction problem (CSP) as follows:

Given n domains $D_1, D_2, \ldots, D_n$ and m constraints $f_1, f_2, \ldots, f_m$, find $x_1, x_2, \ldots, x_n$ such that

$$f_k(x_1, x_2, \ldots, x_n) = 1, \quad 1 \leq k \leq m;$$

$$x_j \in D_j, \quad 1 \leq j \leq n.$$

This problem is only a feasibility problem, and no objective function is defined. Nevertheless, CSPs are an important class of combinatorial optimization problems. Here the functions f_k do not necessarily have closed mathematical forms (for example, functional representations) and can be defined simply by providing the set S described above. A *solution* to a CSP is simply a set of values of the variables such that the values are in the domains of the variables and all of the constraints are satisfied.

Optimization Problems

We have defined a constraint satisfaction problem as a feasibility problem. With regards to optimization, constraint programming systems also allow specification of an objective function. Notationally, we denote the objective function as $g: D_1 \times D_2 \times \ldots \times D_n \rightarrow \Re$, so that at any feasible point to the CSP, the function $g(x_1, x_2, \ldots, x_n)$ can be evaluated. For ease of exposition, we will assume that we are minimizing this objective function. An optimization problem can then be stated as follows:

Minimize $g(x_1, x_2, \ldots, x_n)$

Subject to

$$f_k(x_1, x_2, \ldots, x_n) = 1, \quad 1 \leq k \leq m;$$

$$x_j \in D_j, \quad 1 \leq j \leq n.$$

Simple Examples

To present some simple examples of constraint satisfaction and optimization problems using constraint programming techniques, we will need a language that allows us to describe the decision variables $x_1, x_2, \ldots, x_n$, the constraints $f_1, f_2, \ldots, f_m$, and the objective function g. Because we want to indicate the relationships between constraint programming and mathematical programming, we will use OPL for these representations. We discuss programming a search strategy later. OPL includes a default search strategy for finding a solution to the problem, so we do not always need to describe such a strategy. A later example demonstrates where a search strategy is required to be able to solve the problem. In our presentations of OPL, we include line numbers on the left to facilitate discussion.

Map Coloring

Map coloring is an NP-complete combinatorial optimization problem. Consider a set of countries and a set of colors with a given set of adjacency relationships among the countries. We need to assign the colors to the countries so that no two adjacent countries have the same color. This is an OPL statement for this problem:

```
 1 enum Countries ...;
 2 enum Colors ...;
 3 struct Neighbor {
 4  Countries c1;
 5  Countries c2;
 6 };
 7 setof(Neighbor) neighbors = ...;
 8 var Colors color[Countries];
 9 solve {
10  forall (n in neighbors) {
11   color[n.c1] <> color[n.c2];
12  }
13 };
```

Line 1 specifies the set of countries, and Line 2, the set of colors, with each of these sets given in a data file. Lines 3 through 6 are an OPL construct to define a record, where a record consists of two elements from the set of countries. Line 7 indicates that the set of adjacency relationships (a set called `neighbors` containing records of type `Neighbor`) is also provided as data for the problem. This data is specified as a list of pairs of adjacent countries.

Line 8 contains the decision variables for the problem, with a decision variable `color[j]` for each country `j`, and the value of this decision variable is one of the elements of the set `Colors`. The domains of the variables are elements of a set. Line 9 contains the keyword `solve`, indicating that a constraint satisfaction problem is to be solved. The `forall` statement in Line 10 is an OPL quantifier that says that the constraints on Line 11 will be stated for each member of the set `neighbors`, that is, for each pair of countries that are adjacent. The constraints for this problem, in Lines 10 and 11, are simple to state. The “<>” operator is used to specify that for each member `n` of the set `neighbors`, with each member consisting of a pair of countries `n.c1` and `n.c2`, the color of the first country `n.c1` is different from that of the second country `n.c2` of the pair.

This example illustrates that variables can be set-valued and that constraints can be written as mathematical relations. With a normal mathematical programming representation, it is not possible to write some mathematical relations, such as strict inequalities on continuous variables. In mathematical programming, one normally expresses these relations by introducing additional binary variables and additional constraints into a mixed integer programming representation.

The Stable Marriage Problem

The stable marriage problem is a well-studied problem in the operations research literature. In fact, entire books have been written on the subject, for example, Gusfield and Irving [1989]. Given a set of n women and an equal number of men, we assume that each of the women has provided a preference ordering of the men by assigning each man a unique value from the integers 1 through n. Similarly, we assume that each of the men has provided a preference ordering of the women by assigning each woman a unique value from the integers 1 through n. Each man has a woman he most prefers (a woman he gives a ranking of 1), while each woman also has a man that she prefers (a man she gives a ranking of 1). A stable marriage (for the men) is an assignment of the men to the women so that if a man A prefers another man's (B's) wife, she prefers her current husband B to man A. Similarly, a stable marriage (for the women) is an assignment of the men to the women so that if a woman A prefers another woman's (B's) husband, he prefers his current wife B to woman A. The stable marriage problem is a simplified version of the problem of assigning prospective medical residents to hospitals, that the National Resident Matching Program solves annually.

The conditions for stability are easily stated as a constraint satisfaction problem in OPL:

```
1  enum Women ...;
2  enum Men ...;
3  int rankWomen[Women,Men] = ...;
4  int rankMen[Men,Women] = ...;
5  var Women wife[Men];
6  var Men husband[Women];
7  solve {
8    forall(m in Men)
9     husband[wife[m]] = m;
10   forall(w in Women)
11    wife[husband[w]] = w;
12   forall(m in Men & o in Women)
13    rankMen[m,o] < rankMen[m,wife[m]] =>
14     rankWomen[o,husband[o]] < rankWomen[o,m];
15   forall(w in Women & o in Men)
16    rankWomen[w,o] < rankWomen[w,husband[w]] =>
17     rankMen[o,wife[o]] < rankMen[o,w];
18 };
```

Line 1 specifies the set of women, and Line 2, the men. The elements of these sets can be used to index into arrays, and hence the two-dimensional array `rankWomen` specifies the women's preference rankings of the men, while the two-dimensional array `rankMen` specifies the men's preference rankings of the women. The decision variables for this problem, specified in Lines 5 and 6, are given as two arrays. For each element `m` in the set `Men`, `wife[m]` is an element of the set `Women` that indicates the wife of man `m`. Similarly, for each element `w` in the set `Women`, `husband[w]` is an element of the set `Men` that indicates the husband of wife `w`. The values of the decision variables are specific elements of a set and are not numerically valued.

As in the map-coloring problem, Line 7 indicates that we want to find a solution to a constraint satisfaction problem. Again, we use the `forall` constructor to indicate a group of constraints. On Lines 8 and 9, we state a constraint that specifies that the husband of the wife of man `m` must be man `m`. We state a similar constraint for each woman `w` on Lines 10 and 11. Note that the decision variables `wife[m]` and `husband[w]` are being used to index into the arrays of decision variables `husband[]` and `wife[]`, respectively. This kind of constraint is called an *element* constraint.

Lines 12 through 14, and subsequently Lines 15 through 17, specify the stability relationships, with the rule for the men stated before the rule for the women. The operator "=>" is the logical implication relation, while the "<" is a less-than relation. The constraint in Lines 13 and 14 can be read as, "If `rankMen[m,o] < rankMen[m,wife[m]]`, then it must be the case that `rankWomen[o, husband[o]] < rankWomen[o,m]`." This constraint is satisfied as long as it is not the case that the left-hand side of the implication is true and the right-hand side of the implication is simultaneously false. An alternative representation of this constraint would be to replace the implication relation "=>" with the less-than-or-equal relation "<=", which, in this case, would be an equivalent representation. This is because both the left-hand and right-hand sides are themselves constraints (each using the "<" relation), and these constraints evaluate to 0 or 1 and hence can be numerically compared. This type of composition of constraints is termed a *metaconstraint*, since it is a constraint stated over other constraints.

A Sequencing Problem

A set of blocks in a plant needs to be painted. Each block has a designated color, and we want to sequence the blocks to minimize the cost of the number of times we change the paint color during the sequence. In addition, the blocks are each given an initial position w_i in the sequence and a global interval g such that if a block is painted before its initial position or after the position $w_i + g$, a penalty cost is assessed. This problem is a simplified version of a paint-shop problem solved with constraint programming at Daimler-Chrysler. The following provides an OPL model for this problem:

```
1  int nblocks = ...;
2  enum Colors = ...;
3  range blockrng 1..nblocks;
4  blockrng whenblock[blockrng] = ...;
5  Colors color[blockrng] = ...;
6  int+ OKinterval = ...;
7  int swapcost = ...;
8  assert forall (i in 1..nblocks-1)
         whenblock[i] <= whenblock[i+1];
9
10 setof(int) whencolor[c in Colors] =
11        {i | i in blockrng : color[i]=c};
12 var blockrng position[blockrng];
13 var blockrng whichblock[blockrng];
14 var int colorChanges in card(Colors)
          -1 ..nblocks-1;
15 var int pencost in 0..nblocks*nblocks;
16
17 minimize swapcost*colorChanges + pencost
18 subject to {
19  colorChanges = sum (i in 1..nblocks-1)
20      (color[whichblock[i]]<>color
          [whichblock[i+1]]);
21  pencost = sum (i in blockrng)
22       (maxl(whenblock[i]-position[i],0) +
23        maxl(position[i]-(whenblock[i]
              +OKinterval),0));
24    alldifferent(position);
25    alldifferent(whichblock);
26    forall (i in blockrng) {
27       whichblock[position[i]] = i;
28       position[whichblock[i]] = i;
29    };
30    forall (c in Colors) {
31       forall (k in whencolor[c] : k <>
              whencolor[c].last()) {
32       position[k] < position[whencolor
              [c].next(k)];
33       }
34    }
35 };
```

Lines 1 and 2 specify the number of blocks and the set of paint colors. Line 3 uses the OPL keyword `range` to create a type (`blockrng`) for representing the possible positions (1 through the number of blocks) in which we can place each block. This type can then act as a set of integers, just as the type `Colors` represents the set of colors. Line 4 declares the array `whenblock`, which corresponds to the values w_i. We assume that these values are given in increasing order, and this is checked in Line 8. Line 5 specifies the color of each block, while Line 6 declares the global interval g. Line 7 declares the cost of changing the color in midsequence. Lines 10 and 11 determine, for each possible color, the set of blocks to be painted this color.

The model to solve this problem begins at Line 12. Here, the array `position` indicates the ordinal position of each block, while the array `whichblock` declared in Line 13 indicates which specific block is in each position. Line 14 declares a variable to count the number of times that the color can be changed. A lower bound for this is the number of colors less 1, while an upper bound is the number of blocks less 1. For ease of exposition, the penalty cost is declared as a decision variable in Line 15. Line 17 indicates that the objective function is being minimized. The number of changes in color is determined by comparing the color of adjacent blocks in Lines 19 and 20. As in the previous section, this is done via a metaconstraint, where the inequality `color[whichblock[i]] <> color[whichblock[i+1]]` is evaluated for each pair of adjacent blocks. This inequality has a zero-one value, and these values are summed for each pair to compute the number of changes in color. The penalty costs are computed in Lines 21 through 23. The OPL function `maxl (a, b)` computes the maximum of the arguments. The penalty is then computed by penalizing both earliness and lateness, with no penalty assessed when the condition `whenblock[i] <= position[i] <= whenblock[i]+Okinterval` holds.

Lines 24 and 25 illustrate two examples of *global constraints*. The constraint `alldifferent(position)` indicates that each member of the `position` array should have a different value. In essence, this single constraint on an array of `nblocks` variables is a statement of `nblocks*(nblocks−1)` pairwise inequalities. In this example, because the array `position` has `nblocks` elements, each with an integer value between 1 and `nblocks`, the constraint is equivalent to saying that the array `position` contains an assignment of each block to a unique position. An equivalent representation using mathematical programming techniques would require `nblocks*nblocks` binary variables, and `2*nblocks` constraints. The element constraints in Lines 27 and 28 represent that the arrays `position` and `whenblock` are inverses of each other.

The constraints stated in Lines 30 through 34 help reduce the search space. For each color, the blocks of that color are considered in the order of the values in the `whenblock` array. Because of the structure of the penalties, it is easy to see that blocks of the same color should be ordered in the same order in which they appear in the input data. The constraints are written for each of the blocks of each color. The expression `whencolor[c].next(k)` indicates the block that follows block `k` in the set `whencolor[c]`.

Algorithms for Constraint Programming

Up to now, we have not discussed the algorithm that a constraint programming system uses to determine solutions to constraint satisfaction and optimization problems. In traditional constraint programming systems, the user is required to program a search strategy that indicates how the values of the variables should change so as to find values that satisfy the constraints. In OPL, a default search strategy is used if the user does not provide a search strategy. However, users often program a search strategy to effectively apply constraint programming to solve a problem. We will describe the fundamental algorithm underlying a constraint programming system and then indicate the methodologies used to program search strategies.

Constraint Propagation and Domain Reduction

We defined a constraint as a mathematical function $f(x_1, x_2, \ldots, x_n)$ of the variables. Within this environment, assume there is an underlying mechanism that allows the domains of the variables to be maintained and updated. When a variable's domain is modified, the effects of this modification are then communicated to any constraint that interacts with that variable. This communication is called *con-*

straint propagation. For each constraint, a *domain reduction algorithm* modifies the domains of all the variables in that constraint, given the modification of one of the variables in that constraint. The domain reduction algorithm for a particular kind of constraint discovers inconsistencies among the domains of the variables in that constraint by removing values from the domains of the variables. If the algorithm discovers that a particular variable's domain becomes empty, then it can be determined that the constraint cannot be satisfied, and an earlier choice can be undone. A similar methodology is found in the bound-strengthening algorithms used in modern mathematical programming solvers and discussed by Brearley, Mitra, and Williams [1975]. A crucial difference between the procedures used in mathematical programming presolve implementations and domain reduction algorithms is that in constraint programming, the domains can have holes, while in mathematical programming, domains are intervals.

We can best demonstrate this methodology with an example (Figure 1). Consider two variables x and y, where the domains of each variable are given as $D_x = \{1,2,3,4,\ldots,10\}$ and $D_y = \{1,2,3,4,\ldots,10\}$, and the single constraint $y = 2x$. If we first consider the variable y and this constraint, we know that y must be even, and hence the domain of y can be changed to $D_y = \{2,4,6,8,10\}$. Now, considering the variable x, we see that since $y \leq 10$, it follows that $x \leq 5$, and hence the domain of x can be changed to $D_x = \{1,2,3,4,5\}$. Suppose that we add a constraint of the form (x modulo 2) = 1. This is equivalent to the statement that x is odd. We can then reduce the domain of x to be $D_x = \{1,3,5\}$. Now, reconsidering the original constraint $y = 2x$, we can remove the values of 4 and 8 from the domain of y and obtain $D_y = \{2,6,10\}$.

Some constraint programming systems (for example, ILOG Solver, Oz, and ECLiPSe) allow the programmer to take advantage of existing propagators for built-in constraints that cause domain reductions and allow the programmer to build his or her own propagation and domain reduction schemes for user-defined

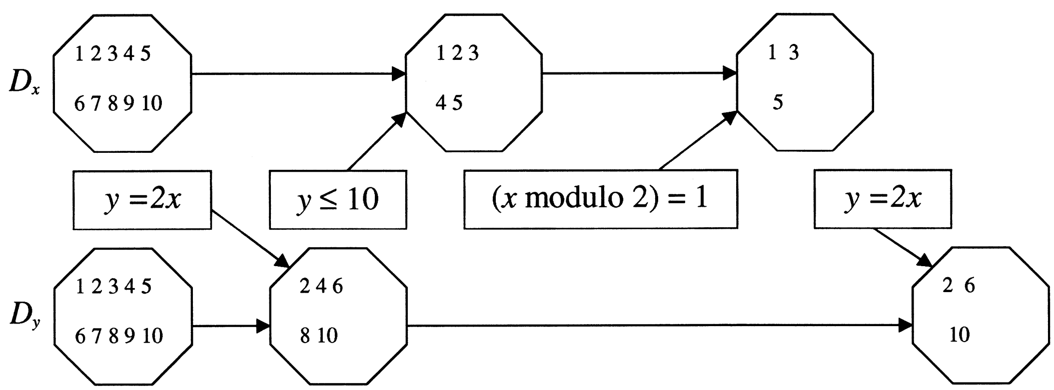

Figure 1: Constraint propagation and domain reduction are used to reduce the domains of the variables x and y. The constraints $y = 2x$, $y \leq 10$, (x modulo 2) = 1, and $y = 2x$ are applied in an order determined by constraint propagation, due to reductions in the domains of each of the variables.

constraints. However, many constraint programming systems (for example, OPL, ILOG Solver, and CHIP) are now strong enough that they provide large libraries of predefined constraints, with associated propagation and domain reduction algorithms, and it is often not necessary to create new constraints with specialized propagation and domain reduction algorithms.

Given a set of variables with their domains and a set of constraints on those variables, a constraint programming system will apply the constraint propagation and domain reduction algorithm in an iterative fashion to make the domains of each variable as small as possible, while making the entire system arc consistent. Given a constraint f_k as stated above and a variable x_j, value $d \in D_j$ is consistent with f_k if at least one assignment of the variables exists such that $x_j = d$ and $f_k = 1$ with respect to that assignment. A constraint is then *arc consistent* if all of the values in the domains of all the variables involved in the constraint are consistent. A constraint system is *arc consistent* if all of the corresponding constraints are arc consistent. The term *arc* is used because the first CSPs were problems with constraints stated on pairs of variables, and hence this system can be viewed as a graph, with nodes corresponding to the variables and arcs corresponding to the constraints. Arc consistency enables the domains of the variables to be reduced while not removing potential solutions to the CSP.

Researchers have developed a number of algorithms to efficiently propagate constraints and reduce domains so as to create systems that are arc consistent. One algorithm, called AC-5, was developed by Van Hentenryck, Deville, and Teng [1992]. Their article is important for constraint programming because it unified the research directions of the constraint satisfaction community and the logic programming community by introducing the concept of developing different domain reduction algorithms for different constraints as implementations of the basic constraint propagation and domain reduction principle.

Constraint Programming Algorithms for Optimization

As compared to linear and mixed-integer programming, a weakness of a constraint programming approach when applied to a problem with an objective function to minimize is that a lower bound may not exist. A lower bound may be available if the expression representing the objective function has a lower bound that can be derived from constraint propagation and domain reduction. This is unlike integer programming, in which a lower bound always exists because of the linear programming relaxation of the problem. Constraint programming systems offer two methods for optimizing problems, called standard and dichotomic search.

The standard search procedure is to first find a feasible solution to the CSP, while ignoring the objective function $g(x_1, x_2, \ldots, x_n)$. Let $y_1, y_2, \ldots, y_n$ represent such a feasible point. The search space can then be pruned by adding the constraint $g(y_1, y_2, \ldots, y_n) > g(x_1, x_2, \ldots, x_n)$ to the system and continuing the search. The added constraint specifies that any new feasible point must have a better objective value than the current point. Propagation of this constraint may cause the domains of the decision variables to be reduced, thus reducing the size of the search space. As the search progresses, new points will have progressively better objective values. The procedure concludes when no feasible point is found. When this happens, the last feasible point can be taken as the optimal solution.

Dichotomic search depends on having a good lower bound L on the objective function $g(x_1, x_2, \ldots, x_n)$. Before optimizing the objective function, a procedure must find an initial feasible point, which determines an upper bound U on the objective function. A dichotomic search procedure is essentially a binary search on the objective function. The procedure computes the midpoint $M = (U + L)/2$ of the two bounds and then solves a CSP by taking the original constraints and adding the constraint $g(x_1, x_2, \ldots, x_n) < M$. If it finds a new feasible point, then it updates the upper bound and continues the search in the same way with a new midpoint M. If it finds the system to be infeasible, then it updates the lower bound, and the search again continues with a new midpoint M. Dichotomic search is effective when the lower bound is strong, because the computation time to prove that a CSP is infeasible can often be large. The use of dichotomic search in cooperation with a linear programming solver may be effective if the linear programming representation can provide a good lower bound. The difference between this procedure and a branch-and-bound procedure for mixed-integer programming is that the dichotomic search stresses the search for feasible solutions, whereas branch-and-bound procedures usually emphasize improving the lower bound.

Programming a Search Strategy

Given a CSP, one can apply the constraint propagation and domain reduction algorithms to reduce the domains of the variables so as to arrive at an arc-consistent system. However, while doing this may determine whether the CSP is infeasible, it does not necessarily find solutions to a CSP. To do this, one must program a search strategy (or use a default search strategy if the constraint programming system provides one). Traditionally, the search facilities that constraint programming systems provide have been based on depth-first search. The root node of the search tree contains the initial values of the variables. At each node, the user programs a *goal*, which is a strategy that breaks the problem into two (or more) parts and decides which part should be evaluated first. A simple strategy might be to pick a variable and to try to set that variable to the different values in the variable's domain. This strategy creates a set of leaves in the search tree and creates what is called a *choice point*, with each leaf corresponding to a specific choice. The goal also orders the leaves amongst themselves within the choice point. In the next level of the tree, the results of the choice made at the leaf are propagated, and the domains are reduced locally in that part of the tree. This will either produce a smaller arc-consistent system or a proof that the choice made for this leaf is not possible. In this case, the system automatically backtracks to the parent and tries other leaves of that parent. The search thus proceeds in a depth-first manner, until it finds a solu-

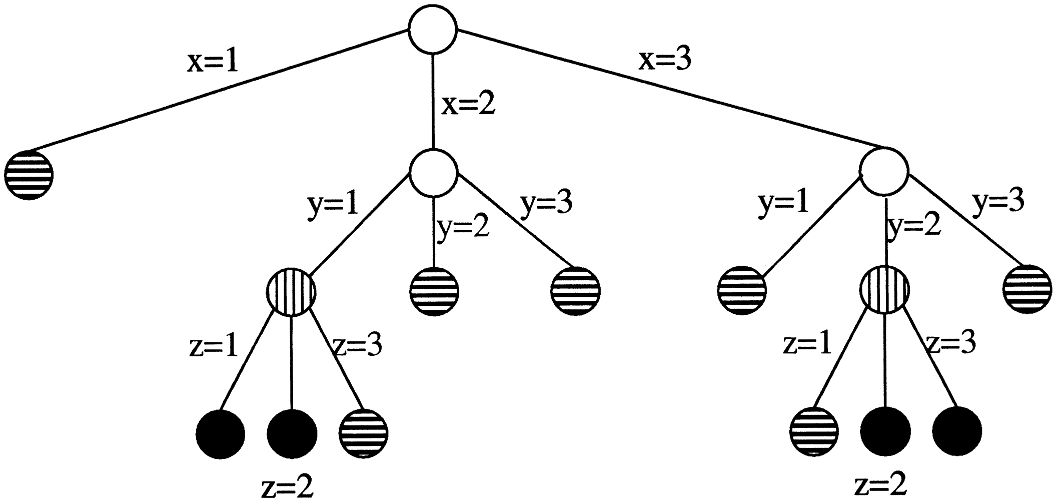

Figure 2: In this diagram, each internal node represents a choice point. The nodes shaded with horizontal lines represent nodes that are never created, because the corresponding values have been removed from the variable's domain. The nodes that are shaded with vertical lines represent nodes that are also never created, because there is a single choice for the decision variable y once the selection for x has been determined. The nodes that are shaded black correspond to solutions that are found. It is worthwhile to consider the node that exists as a result of the choice $x = 3$. After this choice is made, the constraint propagation and domain reduction algorithms automatically remove the values 1 and 3 from the domain of y and there is no need for a choice point corresponding to a choice for y. Since $y = 2$ is the only remaining value, the search for z can begin immediately.

tion at a node low in the tree or until it explores the entire tree, in which case it finds the CSP to be infeasible. The search strategy is enumerative, with constraint propagation and domain reduction employed at each node to help prune the search space.

A simple example will illustrate this idea. Consider the following OPL example, which shows a simple CSP on three variables, each with the same domain.

```
1 var int x in 1..3;
2 var int y in 1..3;
3 var int z in 1..3;
4 solve {
5       x − y = 1;
6       (z = x) \/ (z = y);
7 };
```

The constraint in Line 6 illustrates a logical constraint using the logical or operator (`\/`) indicating that either (or both) of the conditions (`z = x`) or (`z = y`) must hold. A default search strategy for this problem would try the different values for `x, y,` and `z` in order, producing the search tree shown in Figure 2.

ILOG Solver 4.4 and ILOG OPL Studio 2.1 incorporate a recent innovation in constraint programming systems [ILOG 1999], allowing the programmer to use other strategies beyond depth-first search. Constraint programming systems have traditionally used depth-first search because, in viewing constraint programming as a particular computer programming methodology, depth-first search dramatically simplifies memory management issues. ILOG Solver 4.4 and ILOG OPL Studio 2.1 (which uses ILOG's Solver technology)

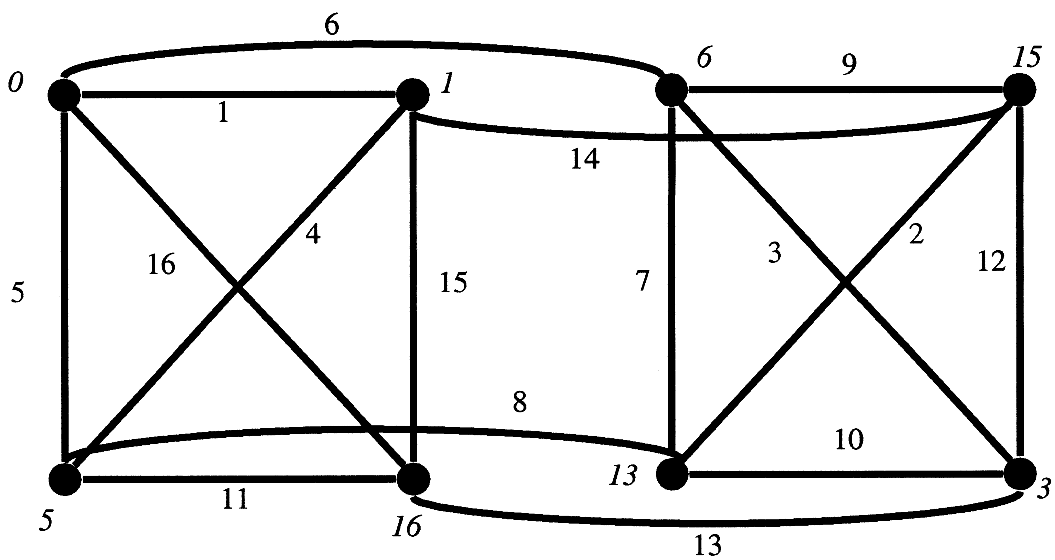

Figure 3: In this graph with a graceful labeling, the numbers in an italic font are the labels for the nodes, while the other numbers are the labels for the edges.

allow the programmer to use best-first search [Nilsson 1971], limited-discrepancy search [Harvey and Ginsberg 1995], depth-bounded-discrepancy search [Walsh 1997], and interleaved depth-first search [Meseguer 1997]. In ILOG Solver, the basic idea is that the user programs *node evaluators*, *search selectors*, and *search limits*. Node evaluators contain code that looks at each open node in the search tree and chooses one to explore next. Search selectors order the different choices within a node, and search limits allow the user to terminate the search after it reaches some global limit (for example, time or node count). With these basic constructs in place, one can easily program any search strategy that systematically searches the entire search space by choosing nodes to explore (that is, programming node evaluators), by dividing the search space at nodes (that is, programming goals and creating choice points), and by picking the choice to evaluate next within a specific node (that is, programming search selectors). Constraint programming systems provide a framework for describing enumeration strategies for solving search problems in combinatorial optimization.

An Example of Search: Graceful Graphs

A graph $G=(V,E)$ with $n=|V|$ nodes and $m=|E|$ edges is said to be *graceful* if there are unique node labels $f{:}V \rightarrow \{0,1,2,\ldots,m\}$ and unique edge labels $g{:}E \rightarrow \{1,2,\ldots,m\}$ such that $g(i,j)=|f(i)-f(j)|$ for each edge $e \in E$ with $e=(i,j)$ (Figure 3). Graceful graphs have applications in radio astronomy and cryptography.

An OPL solution to the problem of finding a graceful labeling of a graph follows:

```
int numnodes = ...;
range Nodes 1..numnodes;
struct Edge {
    Nodes i;
    Nodes j;
};
setof(Edge) edges = ...;
int numedges = card(edges);
range Labels 0..numedges;
```

```
10 var Labels nl[Nodes];
11 var Labels el[edges];
12 solve {
13     alldifferent(nl);
14     alldifferent(el);
15     forall (e in edges) {
16        el[e]=abs(nl[e.i]−nl[e.j]);
17        el[e]>0;
18     }
19 };
20 search {
21     generate(nl);
22 };
```

Line 1 declares that the number of nodes is input data, while Line 2 declares a type to represent the set of nodes. Lines 3 through 6 declare a type to hold the pairs of edges, while Line 7 declares the set of edges as input data. A type `Labels` is created at Line 9 to represent the potential labels of the nodes and edges. Lines 10 and 11 declare the arrays `nl` and `el` for the labels of the nodes and edges, respectively. The constraints for the problem are stated in Lines 12 through 19. Lines 13 and 14 use the global constraint `alldifferent` to indicate that all the node labels must be different as well as the edge labels. Lines 15 through 18 indicate the relationship between the edge labels and the node labels. Because all node labels must be distinct, the constraint in Line 17 indicates that the zero value is not possible for the edge labels.

Lines 20 through 22 indicate a very simple search procedure for solving this problem. The statement `generate(nl)` indicates that the different possible values for the array `nl` should be generated in a nondeterministic way by choosing the variable with the smallest domain size and instantiating that variable to the smallest value in its domain. After this selection is made, constraint propagation will cause values to be removed from the domains of other variables. A new variable is then chosen and a value is instantiated for it. As the search progresses, a new variable is chosen, always using the smallest domain size as the metric. It should be noted that without the declaration of this procedure, it is not possible to solve the example in Figure 3. In fact, until constraint programming was applied to this particular example, it was not known if the graph in this figure had a graceful labeling.

An Example of Search: A Warehouse Location Problem

Consider the problem of assigning stores to warehouses while simultaneously deciding which warehouses to open. The data that is given is the cost of assigning each store to each warehouse, a fixed cost that is incurred when the warehouse is opened, and the capacity (in terms of number of stores) of each potential warehouse. An OPL model for this problem using constraint programming constructs and a search strategy for solving the problem follows:

```
int fixed = ...;
int nbStores = ...;
enum Warehouses ...;
range Stores 1..nbStores;
int capacity[Warehouses] = ...;
int supplyCost[Stores,Warehouses] = ...;
int maxCost = max(s in Stores, w in
            Warehouses) supplyCost[s, w];
Warehouses wlist[w in Warehouses] = w;

var int open[Warehouses] in 0..1;
var Warehouses supplier[Stores];
var int cost[Stores] in 0..maxCost;

minimize
     sum(s in Stores) cost[s] + sum(w in
            Warehouses) fixed * open[w]
subject to {
      forall(s in Stores)
         cost[s] = supplyCost[s, supplier[s]];
      forall (s in Stores)
         open[supplier[s]] = 1;
      atmost(capacity, wlist, supplier);
};

search {
```

```
25     forall(s in Stores ordered by decreasing
                  regretdmin(cost[s]))
26     tryall(w in Warehouses
27      ordered by increasing supplyCost[s,w])
28     supplier[s] = w;
29 };
```

Lines 1 through 6 declare the data for the problem and use the OPL keyword `range` to create a type (`Stores`) to correspond to the range of integers 1 through the number of stores. In Line 7, the maximum cost of assigning any store to any warehouse is computed. Line 8 computes the list of warehouses as an array for use in the `atmost` constraint in Line 21.

Lines 10 through 12 contain the decision variables for the problem. Line 10 declares an array of binary variables to indicate whether a particular warehouse is open. Line 11 declares an array that will contain which warehouse is assigned to a particular store. For any store `s`, `supplier[s]` is an element of the set `Warehouses`. We also will create an array of decision variables `cost[]` that will contain the actual cost of assigning a particular store to its warehouse. This will be used in the search strategy described in Lines 24 through 29.

Lines 14 and 15 indicate that we wish to minimize a cost function. Lines 17 and 18 compute the cost of assigning a particular store by looking up the cost in the data array `supplyCost[]`. Lines 19 and 20 describe the constraints that require that, if a store is assigned to a warehouse, the warehouse must be open. Finally, the constraint expressed in Line 21 is a particular kind of global constraint, described earlier. The `atmost` constraint in this particular case says that for each value `wlist[w]` (which, in this case, is each element `w` of the set `Warehouses`), the number of times that this value may appear in the array `supplier` is at most `capacity[w]`. Effectively, this single constraint is counting the number of times each warehouse is used and placing a bound on that count.

Lines 24 through 29 describe a search strategy for solving this problem. The basic idea behind the search strategy is to order the stores via some merit function and then, for each store, to try different warehouses to which it can be assigned, after ordering the warehouses by a different merit function. The statement in Line 25 orders the stores by what is known as the minimal-regret heuristic. For a particular store `s`, we can easily see that the domain of possible values of the variable `cost[s]` is initialized to be the values in the data matrix `supplyCost[s,w]` for each possible warehouse `w`. Hence, if there were five warehouses, the initial domain of `cost[s]` would have five values. As the search progresses toward a solution, values are eliminated from the domain of this variable. At any particular time, we can consider the two lowest costs in the domain of `cost[s]`. The value `regretdmin(cost[s])` is defined as the difference between these two values. The abbreviation `dmin` corresponds to domain minimum and the abbreviation `regret` indicates the regret, or the cost of switching from the minimum-cost warehouse for that store to the next-higher-cost warehouse. The statement in Line 25 dynamically orders the stores by this value, that is, it orders the stores by ranking first the store that would incur the highest cost of switching from the lowest-cost warehouse to the next-lowest-cost warehouse. Lines 26 and 27 then order the warehouses for that store by ranking first the warehouse with the lowest cost. In combination with Line 28, the `tryall` instruction in Line 26 says that the system should try to assign each warehouse in turn to this store. After one store is assigned, the next store that is ranked according to the minimal-regret heuristic is assigned to its warehouse. If this assignment is found to be infeasible, the system will backtrack and try a different warehouse for the last store that was assigned. Since no search strategy is specified for the variables `cost` and `open`, the OPL system will use a default search strategy for those variables after the supplier variables have been assigned. In this particular case, the values of `cost` and `open` are determined by the constraints in Lines 18 and 20, respectively, so no specific search strategy for those variables is required.

This search strategy uses knowledge about the problem to guide the search. First, it chooses to assign stores to warehouses and then to open the warehouses. By ranking the stores according to the cost of switching and by ranking the warehouses by the cost of assigning the warehouses to a fixed store, the search strategy prunes the search space by trying the lower-cost warehouses first in the search for a solution. In one particular instance, without this search strategy, ILOG OPL Studio 3.0 needed 1,894 choice points to find a solution. With this search strategy, ILOG OPL Studio 3.0 needed only 147 choice points.

Connections to Integer Programming

We have emphasized how constraint programming can be applied to combinatorial optimization problems. The search strategies used in constraint programming are related to those used when solving mixed-integer-programming problems via branch-and-bound procedures. Furthermore, the problems that are solved often have integer programming representations.

For those familiar with integer programming, the concept of search strategies we have described should seem familiar. In fact, branch and bound, which is an enumerative search strategy, has been used to solve integer programs since the mid-1960s. Lawler and Wood [1966] give an early survey, while Garfinkel and Nemhauser [1972], in their classic text, describe branch and bound in the context of an enumerative procedure. Nemhauser and Wolsey [1988] provide a more recent discussion. In systems developed for integer programming, users are often given the option of choosing a variable selection strategy and a node selection strategy. These are clearly equivalent to the search selectors and node evaluators we described.

A constraint programming framework extends the basic branch-and-bound procedures implemented in typical mixed-integer programming solvers in two fundamental ways. First, in most implementations of branch-and-bound procedures for mixed-integer programming, the implementation creates two branches at each node after a variable x with a fractional value v has been chosen to branch on. The implementation then divides the search space into two parts by creating a choice point based on the two choices of $(x \leq \lfloor v \rfloor)$ and $(x \geq \lceil v \rceil)$. In the constraint programming framework, the choices that are created can be any set of constraints that divides the search space. For example, given two integer variables x_1 and x_2, one could create a choice point consisting of the three choices $(x_1 < x_2)$, $(x_1 > x_2)$, $(x_1 = x_2)$.

The second way that a constraint programming framework extends the basic branch-and-bound procedures is with respect to the variable selection strategy. In most branch-and-bound implementations, the variable selection strategy uses no knowledge about the model of the problem to make the choice of variable to branch on. The integer program is treated in its matrix form, and different heuristics are used to choose the variable to branch on based on the solution of the linear programming relaxation that is solved at each node. In a constraint programming approach, the user specifies the branching strategy in terms of the formulation of the problem. Because a constraint program is a computer program, the decision variables of the problem can be treated as computer programming variables, and one programs a strategy using the language of the problem formulation. Hence, to effectively apply constraint programming techniques, one uses problem-specific knowledge to help guide the search strategy so as to efficiently find a solution. In this way, a constraint programming system, when combined with a linear programming optimizer, can be viewed as a framework that allows users to program problem-specific branch-and-bound search strategies for solving mixed-integer programming problems by using the same programming objects for declaring the decision variables and for programming the search strategies. Combinations of linear programming and constraint programming have appeared in Prolog III [Colmerauer 1990], CLP(R) [Jaffar and Lassez 1987], CHIP [Dincbas et al. 1988], ILOG Solver and ILOG Planner [ILOG 1999] and ECLiPSe [Wallace, Novello, and Schimpf 1997].

A number of branch-and-bound implementations for mixed-integer programming allow users to program custom search strategies, including MINTO [Nemhauser, Savelsbergh, and Sigismondi 1994], IBM's OSL [1990], Dash Associates XPRESS-MP [2000], and ILOG CPLEX [2000]. In particular, MINTO allows users to divide the search space into more than one part at each node. However, the crucial difference between constraint programming systems and these mixed-integer programming branch-and-bound solvers is that the mixed-integer programming systems require the users to specify their search strategy in terms of a matrix formulation of the problem, whereas a constraint programming system uses a single programming language for both modeling the problem to be solved and for specifying a search strategy to solve the problem. The key point is that the decision variables of an optimization problem can be treated as programming language variables within a computer programming environment.

Contrasting Formulations

Earlier we gave an example of solving a warehouse location problem using constraint programming techniques. It is worth contrasting this formulation with a pure integer programming formulation:

```
int fixed = ...;
int nbStores = ...;
enum Warehouses ...;
range Stores1..nbStores;
int capacity[Warehouses] = ...;
int supplyCost[Stores,Warehouses] = ...;

var int open[Warehouses] in 0..1;
```

```
 9 var int supply[Stores,Warehouses] in 0..1;
10
11 minimize
12   sum(w in Warehouses) fixed*open [w] +
13   sum(w in Warehouses, s in Stores)
             supply Cost [s,w] * supply [s,w]
14 subject to {
15      forall(s in Stores)
16         sum(w in Warehouses) supply [s, w] = 1;
17      forall(w in Warehouses, s in Stores)
18        supply [s,w] <= open[w];
19      forall(w in Warehouses)
20         sum(s in Stores) supply[s,w]
                       <= capacity[w];
21 };
```

Lines 1 through 6 are identical to the constraint programming formulation earlier and specify the data for the problem. The meaning of the array `open[]` is also the same. For each potential pair of a store `s` and a warehouse `w`, the Boolean variable `supply[s,w]` indicates whether store `s` is assigned to warehouse `w`. The objective function is stated in Lines 12 through 13. The cost of assigning store `s`, which was represented by a decision variable in the constraint programming formulation, is implicitly computed in the expression in Line 13. Lines 15 and 16 specify the constraint that each store must be assigned to exactly one warehouse. Lines 17 and 18 specify that a store can be assigned only to an open warehouse, while Lines 19 and 20 specify the constraint that limits the number of stores that can be assigned to any particular warehouse.

The constraint programming formulation uses a linear number of variables with larger domains to describe which stores are assigned to which warehouses, while the integer programming formulation uses a quadratic number of binary variables. The constraint that a store can be assigned to exactly one warehouse is implicit in the constraint programming formulation because there is a decision variable `supplier` that indicates the specific warehouse that is assigned. In the integer programming formulation, constraints explicitly enforce this restriction. In the integer programming formulation, the assignment of stores to open warehouses is enforced by an inequality, while the constraint programming formulation enforces this restriction via an element constraint. Finally, the constraint programming formulation uses a global constraint to enforce the restriction on the number of stores per warehouse, while this restriction is enforced with a set of linear constraints in the integer programming formulation.

Is one formulation better than the other is? Making a direct comparison is difficult because mixed-integer programming solvers like CPLEX have advanced techniques, such as cut generation and presolve reductions, that improve the performance of the MIP optimizer. In addition, the performance of the constraint programming algorithms sometimes depends on the underlying data for the problem as well as the effectiveness of the search strategy. An additional consideration is whether one is interested in obtaining a proof of optimality or just interested in a good feasible solution within a certain time limit. We need more research to better understand how to make this kind of comparison.

Hybrid Strategies

One of the exciting avenues of research is to explore how constraint programming and traditional mathematical programming approaches can be used together to solve difficult problems. We indicated how a constraint programming system can serve as a framework for programming a branch-and-bound procedure for integer programming that takes advantage of the problem structure to influence the variable and node selection process. There are other avenues in which the two approaches can cooperate. In the first, we consider stating formulations that contain a mixture of linear and "not-linear" constraints. In the second, we consider decomposing a problem to use constraint programming to solve one part of the problem and mathematical programming to solve a second part.

A Hybrid Formulation of the Warehouse Location Problem

Another way of solving the warehouse location problem is to combine the two formulations. In the following OPL model, we omit the data section and the search strategy (which are the same as in the previous formulations) and just show the decision variables and the constraints.

```
 1 var int open[Warehouses] in 0..1;
 2 var int supply[Stores,Warehouses] in 0..1;
 3 var Warehouses supplier[Stores];
 4 var int cost[Stores] in 0..maxCost;
 5
 6 minimize with linear relaxation
 7   sum (w in Warehouses) fixed*open[w] +
 8       sum(w in Warehouses, s in Stores)
            supplyCost[s,w]*supply[s,w]
 9 subject to {
10   forall(s in Stores)
11       sum(w in Warehouses) supply[s,w] = 1;
12   forall(w in Warehouses, s in Stores)
13       supply[s,w] <= open[w];
14   forall (w in Warehouses)
15    sum(s in Stores) supply[s,w]<=capacity[w];
16
17   forall(s in Stores)
18       cost[s] = supplyCost[s,supplier[s]];
19   forall(s in Stores)
20       open[supplier[s]] = 1;
21   atmost(capacity, wlist, supplier);
22
23   forall(s in Stores)
24       supply[s,supplier[s]] = 1;
25   forall(s in Stores)
26       cost[s] = sum(w in Warehouses)
               supplyCost[s,w]*supply[s,w]
27 };
```

Lines 1 through 4 combine the variables of the two formulations and have the same meaning. In Line 6, the phrase "`with linear relaxation`" indicates that OPL should solve the problem by extracting the linear constraints and using the objective function on the linear relaxation of those constraints to determine a lower bound that can be used to prove optimality of the procedure. (We do not use dichotomic search here.) Lines 7 through 8 are the same objective function. Lines 10 through 15 are the constraints from the linear formulation, while Lines 17 through 21 are the constraints from the constraint programming formulation. Lines 23 through 26 are the constraints that link the two formulations. Lines 23 and 24 relate the `supply` and `supplier` variables, while Lines 25 and 26 determine the value of the `cost` variable in terms of the `supply` variables. The additional constraints help the combined formulation to prune the search space and reduce the overall solution time. For example, we can solve a simple five-warehouse, five-store problem using the pure constraint programming formulation with 147 choice points, while we can solve the same problem using the hybrid formulation with 53 choice points. For this particular data set, we can solve all three formulations in a fraction of a second, making their comparison difficult.

With the advances in available technology, exploring these kinds of mixed formulations is now possible. Such techniques might prove to be effective solution strategies for hard combinatorial optimization problems that benefit from multiple representations. Jain and Grossman [2000] have investigated these concepts for some machine-scheduling applications.

Column Generation and Crew Scheduling

Constraint programming can be used when implementing column-generation approaches for solving different kinds of combinatorial optimization problems. The classic example is the cutting stock problem, in which one solves knapsack problems to generate the possible patterns to use and then solves linear programs to determine how many cuts of each pattern to use. The dual solutions to each linear program change the cost structure of the knapsack problem, which one solves to generate a new column of the linear program.

Another example is crew scheduling. For a number of years, people have solved crew scheduling problems by writing computer programs to generate the potential pairings of crews to flights. The programs have to generate pairings, corresponding to a sequence of flights for a single crew, that are low in cost and satisfy the complex duty rules included in volumes of carrier regulations. The typical approach used in the past, which is still used today, is a *generate-and-test* approach: One generates a complete pairing and then tests the feasibility of this pairing against all of the rules programmed into the system. After generating a suitable number of pairings, one solves a set-partitioning problem by letting each pairing correspond to one column of the set-partitioning problem.

Constraint programming can be used to generate the pairings. Given a set of flights and a flight schedule, one can declare variables that correspond to the sequence of flights covered by one pairing. One can then state constraints that dictate the rules about sequences of flights, how many hours can be in a total sequence of flights, the required time between flights, and so forth. One can then carry out a search procedure to generate potential flight sequences. As the search proceeds, the underlying constraint propagation and domain reduction algorithms prune the search space. This becomes more of a *test-and-generate* method, where the constraints that define possible pairings are consistently used to help guide the search for possible feasible solutions. After generating a suitable set of pairings, one can use them as columns for a set-partitioning problem which is solved by an integer programming solver. An example of this approach can be found on the ILOG Web site in the OPL model library at `http://www.ilog.com/products/oplstudio/`.

Many operations research applications incorporate various kinds of enumeration strategies that are usually embedded in complex computer programs. Constraint programming provides an attractive alternative for implementing these kinds of enumeration schemes. We prefer to call this *constrained enumeration*.

Constraint-Based Scheduling

Another interesting application of constraint programming is in scheduling problems. We define a scheduling problem as a problem of determining a sequence of activities with given precedence relationships, subject to constraints on resources that the activities compete for. For example, the classic job shop scheduling problem, in which a set of jobs consisting of sequential tasks must be scheduled on machines, fits into this framework. To support this class of problems, many constraint programming systems extend their frameworks to directly represent these problems. Following is the job shop problem stated in OPL:

```
1  int nbMachines = ...;
2  range Machines 1..nbMachines;
3  int nbJobs = ...;
4  range Jobs 1..nbJobs;
5  int nbTasks = ...;
6  range Tasks 1..nbTasks;
7
8  Machines resource[Jobs,Tasks] = ...;
9  int+ duration[Jobs,Tasks] = ...;
10 int totalDuration = sum(j in Jobs, t in Tasks)
                       duration[j,t];
11
12 scheduleHorizon = totalDuration;
13 Activity task[j in Jobs, t in Tasks]
                       (duration[j,t]);
14 Activity makespan(0);
15
16 UnaryResource tool[Machines];
17
18 minimize
19    makespan.end
20 subject to {
21    forall(j in Jobs)
22         task[j, nbTasks] precedes makespan;
23
24    forall(j in Jobs)
25         forall(t in 1..nbTasks-1)
26            task[j,t] precedes task[j,t + 1];
27
28      forall(j in Jobs)
29         forall(t in Tasks)
30            task[j,t] requires
                 tool[resource[j,t]];
31 };
```

The first nine lines define the input data of the problem, consisting of the number of machines, the number of jobs, and the number of tasks. We use the OPL keyword `range` to create a type to correspond to an integer range. The array `resource`, declared in Line 8, is input data consisting of the identity of the machine that is needed to do a particular task within a job. Line 9 declares the array `duration` that is the time required to execute each task within each job. Line 10 computes the maximum duration of the entire schedule, which is used in Line 12 to set the schedule horizon for OPL. Line 13 declares a decision variable `task[j,t]` for each job `j` and task `t` that is an activity. By default, the keyword `Activity` implicitly indicates a decision variable. An activity consists of three parts—a start time, an end time, and a duration. In the declaration given here, the durations of each activity are given as data. When an activity is declared, the constraint

```
task[j,t].start + task[j,t].duration
  = task[j,t].end
```

is automatically included in the system. Line 14 declares an activity `makespan` of duration 0 that will be the final activity of the schedule. Line 16 declares the machines to be unary resources. A unary resource is also a decision variable, and we need to decide what activity will be assigned to that resource at any particular time.

The problem is then stated in Lines 18 through 31. Lines 18 and 19 indicate that our objective is to finish the last activity as soon as possible. Lines 21 and 22 indicate that the last task of each job should precede this final activity. Lines 24 through 26 indicate the precedence order of the activities within each job. The word `precedes` is a keyword of the language. In the case of Line 26, the constraint is internally translated to the constraint

```
task[j,t].end <=
               task[j,t+1].start
```

Finally, Lines 28 through 30 indicate the requirement relationship between activities and the machines by using the `requires` keyword. The declaration of the set of requirements causes the creation of a set of disjunctive constraints. For a particular machine `m` described by a unary resource `tool[m]`, let `task[j1,t1]` and `task[j2,t2]` be two activities that require that machine `m`. In other words, suppose that the data is given such that `resource[j1,t1]=resource[j2,t2]=m`. Then the following disjunctive constraint, created automatically by the system, describes the fact that the two activities cannot be scheduled at the same time:

```
task[j1,t1].end<=task[j2,t2].start\/
task[j2,t2].end<=task[j1,t1].start
```

Here, the symbol "\/" means "or," and the constraint states that either `task[j1,t1]` precedes `task[j2,t2]` or vice versa.

In practice, the kinds of scheduling problems that are solved using constraint programming technologies have more varied characteristics than the simple job shop scheduling problem. Activities can be breakable, allowing the representation of activities that must be inactive on weekends. Resources can be single machines, discrete resources such as a budget, or reservoirs that are both consumed and produced by different activities. Constraint programming is an effective technology for solving these kinds of problems for a number of reasons. First, the nature and strength of the constraint propagation and domain reduction algorithms developed specifically for scheduling help to immediately prune the search space and determine bounds on the start and end times of activities. Second, in practice one does not necessarily need to find a provably optimal solution to such problems but to quickly find a good feasible solution. The architecture of a constraint programming system is suited to finding such solutions, and problems with millions of activities have been successfully solved in this way.

Conclusions

Our experience indicates that constraint programming is better than integer programming in applications that concern sequencing and scheduling, and for problems in which an integer programming formulation contains much symmetry. In addition, strict feasibility problems that are in essence constraint satisfaction problems are good candidates for applying constraint programming techniques. Integer programming seems to be superior for problems in which the linear programming relaxations provide strong bounds for the objective function. Hybrid techniques are relatively new, although Jain and Grossman [2000] describe their effective use in some machine-scheduling applications.

Sessions at recent INFORMS meetings give evidence of growing interest in how constraint programming technologies and mathematical programming approaches can complement each other. Because of the introduction of such languages as OPL and such systems as ECLiPSe, we can easily explore alternative and hybrid approaches to solving difficult problems. We think that these explorations will continue and that additional difficult industrial problems will be solved using combinations of the two techniques. We hope that operations research professionals will become as familiar with constraint programming as they are today with linear programming.

Acknowledgments

We would like to acknowledge the comments of the two anonymous referees as well as the copyeditor. Their comments greatly improved the presentation of this paper.

References

Bisschop, Johannes and Meeraus, Alexander 1982, "On the development of a general algebraic modeling system in a strategic planning environment," *Mathematical Programming Study 20*, pp. 1–29.

Brearley, A. L.; Mitra, Gautam; and Williams, H. P. 1975, "Analysis of mathematical programming problems prior to applying the simplex algorithm," *Mathematical Programming*, Vol. 8, No. 1, pp. 54–83.

Caseau, Yves and Laburthe, F. 1995, "The Claire documentation," LIENS Report 96-15, Ecole Normale Supérieure, Paris.

Colmerauer, Alain 1990, "An introduction to PROLOG III," *Communications of the ACM*, Vol. 33, No. 7, pp. 70–90.

Dantzig, George B. 1948, "Programming in a linear structure," Comptroller, USAF, Washington, DC, February.

Dantzig, George B. 1949, "Programming of interdependent activities, II, mathematical model," *Econometrica*, Vol. 17, Nos. 3 and 4 (July–October), pp. 200–211.

Dantzig, George B. 1963, *Linear Programming and Extensions*, Princeton University Press, Princeton, New Jersey.

Dantzig, George B. and Thapa, Mukund N. 1997, *Linear Programming 1: Introduction*, Springer, New York.

Dash Associates 2000, *XPRESS-MP Optimiser Subroutine Library XOSL Reference Manual (Release 12)*, Dash Associates, Blisworth, United Kingdom.

Dincbas, Mehmet; Van Hentenryck, Pascal; Simonis, Helmut; Aggoun, Abderrahmane; Graf, T.; and Berthier, F. 1988, "The constraint logic programming language CHIP," *Proceedings of the International Conference on Fifth Generation Computer Systems*, Tokyo, Japan, December.

Fourer, Robert; Gay, David M.; and Kernighan, Brian W. 1993, *AMPL: A Modeling Language for Mathematical Programming*, Scientific Press, San Francisco, California.

Garfinkel, Robert S. and Nemhauser, George L. 1972, *Integer Programming*, John Wiley and Sons, New York.

Gusfield, Dan and Irving, Robert W. 1989, *The Stable Marriage Problem: Structure and Algorithms*, MIT Press, Cambridge, Massachusetts.

Harvey, William D. and Ginsberg, Matthew L. 1995, "Limited discrepancy search," *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, Vol. 1, pp. 607–613.

Hooker, John 2000, *Logic-Based Methods for Optimization: Combining Optimization and Constraint Satisfaction*, John Wiley and Sons, New York.

IBM 1990, *Optimization Subroutine Library Guide and Reference (Release 1)*, IBM, Kingston, New York.

ILOG 1999, *ILOG Solver 4.4 Users Manual*, ILOG, Gentilly, France.

ILOG 2000, *ILOG CPLEX 7.0 Reference Manual*, ILOG, Gentilly, France.

Jaffar, Joxan and Lassez, Jean-Louis 1987, "Constraint logic programming," *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages*, Munich, Germany, pp. 111–119.

Jain, Vipul and Grossman, Ignacio 2000, "Algorithms for hybrid MILP/CP models for a class of optimization problems," working paper, Department of Chemical Engineering, Carnegie Mellon University.

Knuth, Donald E. 1968, *Fundamental Algorithms, The Art of Computer Programming*, Vol. 1, second edition, Addison-Wesley, Reading, Massachusetts.

Lauriere, Jean-Louis 1978, "A language and a program for stating and solving combinatorial problems," *Artificial Intelligence*, Vol. 10, No. 1, pp. 29–127.

Lawler, Eugene L. and Wood, D. E. 1966, "Branch-and-bound methods: A survey," *Operations Research*, Vol. 14, No. 4, pp. 699–719.

Mackworth, Alan K. 1977, "Consistency in networks of relations," *Artificial Intelligence*, Vol. 8, No. 1, pp. 99–118.

Marriott, Kim and Stuckey, Peter J. 1999, *Programming with Constraints: An Introduction*, MIT Press, Cambridge, Massachusetts.

Meseguer, Pedro 1997, "Interleaved depth-first search," *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, Vol. 2, pp. 1382–1387.

Nemhauser, George W. and Wolsey, Laurence A. 1988, *Integer Programming*, John Wiley and Sons, New York.

Nemhauser, George W.; Savelsbergh, Martin W. P.; and Sigismondi, Gabriele C. 1994, "MINTO, a Mixed INTeger Optimizer," *Operations Research Letters*, Vol. 15, No. 1, pp. 47–58.

Nilsson, Nils J. 1971, *Problem Solving Methods in Artificial Intelligence*, McGraw-Hill, New York.

Puget, Jean-François 1992, "Pecos: A high level constraint programming language," *Proceedings of the Singapore International Conference on Intelligent Systems (SPICIS)*, Singapore, pp. 137–142.

Puget, Jean-François and Leconte, Michel 1995, "Beyond the glass box: Constraints as objects," *Logic Programming: Proceedings of the 1995 International Symposium*, ed. John Lloyd, MIT Press, Cambridge, Massachusetts, pp. 513–527.

Siskind, Jeffrey M. and McAllester, David A. 1993, "Nondeterministic Lisp as a substrait for constraint logic programming," *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-93)*, pp. 133–138.

Smolka, G. 1993, "Survey of Oz: A higher-order concurrent constraint language," *Proceedings of ICLP '93: Post-Conference Workshop of Concurrent Constraint Programming*, Budapest, Hungary, June 24–25, 1993.

Van Hentenryck, Pascal 1989, *Constraint Satisfaction in Logic Programming*, MIT Press, Cambridge, Massachusetts.

Van Hentenryck, Pascal 1999, *The OPL Optimization Programming Language*, MIT Press, Cambridge, Massachusetts.

Van Hentenryck, Pascal; Deville, Yves; and Teng, C. M. 1992, "A generic arc-consistency algorithm and its specializations," *Artificial Intelligence*, Vol. 57, No. 2, pp. 291–321.

Wallace, Mark; Novello, Stefano; and Schimpf, Joachim 1997, "ECLiPSe: A platform for constraint logic programming," *ICL Systems Journal*, Vol. 12, No. 1, pp. 159–200.

Walsh, Toby 1997, "Depth-bounded discrepancy search," *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, Vol. 2, pp. 1388–1393.

Williams, H. P. 1999, *Model Building in Mathematical Programming*, fourth edition, John Wiley and Sons, New York.