Program ≠ Program: Constraint Programming and its Relationship to Mathematical Programming

Irvin J. Lustig, ILOG, 1080 Linda Vista Ave., Mountain View, CA 94043 USA

Jean-François Puget, ILOG, 9 rue Verdun, BP 85, Gentilly Cedex, France

Original Version: September, 1999. Revised: December, 2000.

Abstract

Arising from research in the computer science community, constraint programming is a relatively new technique for solving optimization problems. For those familiar with mathematical programming, there are a number of language barriers that exist that make it difficult to understand the concepts of constraint programming. This paper, which assumes a minimal background in linear and integer programming, is a short tutorial on constraint programming, and explains how it relates to familiar mathematical programming concepts. We also indicate how constraint programming and mathematical programming technologies are complementary.

1 Introduction

The simplex method for linear programming was invented by George Dantzig in 1947, and described 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-1980's, constraint programming was developed 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 community as a new and sometimes better way of solving certain kinds of optimization problems. This paper serves as an introduction to constraint programming for those readers familiar with traditional mathematical programming.

In Section 2 we discuss the origins of mathematical programming and constraint programming, and contrast the meaning of the word "programming" in both contexts. In Section 3 we give examples of constraint programming formulations for constraint satisfaction problems and optimization problems after providing a formal definition of each. Section 4 discusses the fundamental algorithms of constraint programming, while Section 5 contrasts constraint programming and integer programming as applied to the same problem. Section 6 discusses ways that constraint programming can be combined with mathematical programming, followed by a discussion in Section 7 on scheduling, an important application of constraint programming. Section 8 concludes with a summary and an outlook towards the future.

2 The word "Programming"

For those familiar with mathematical programming, one of the confusions with regards to understanding constraint programming is the fact that the names of both areas share the word "programming." In fact, the use of this word is different for each discipline. It is worth reviewing the origins of both areas to understand this difference.

2.1 "Programming" in Mathematical Programming

The field of mathematical programming arose from its roots in linear programming. In Dantzig's seminal textbook [Dantzig 1963], linear programming is introduced by describing a few different planning problems. Dantzig writes:

> "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 to Dantzig to use the term "linear programming" as opposed to "programming in a linear structure", and Professor Al Tucker then suggested to Dantzig to call the "linear programming problem" a "linear program." [Dantzig 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.

2.2 "Programming" in Constraint Programming

Constraint programming is often called constraint logic programming, and originates in the artificial intelligence literature in the computer science community. Here, the word "programming" refers to computer programming. Knuth [1968] 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 of operations of the computer, and hence the common concept of a "plan" is shared with the origins of linear programming. With respect to constraint programming, it 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. 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] 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 specifies 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 1970's, with the advent of 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 1980's, 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 was termed *finite domain constraints.*

In the 1990's, constraint logic programming, based on Prolog, was transformed to constraint programming, by providing constraint programming features in general purpose programming languages. Examples include Pecos for Lisp [Puget, 1992], ILOG Solver for C++ [ILOG 1999], and Screamer for Common Lisp [Siskind and McAllester 1993]. This allowed the development of 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 the design of such languages, an axiom of their development has been that they provide completeness with respect to being languages for doing computer programming, and hence allow constraint solving algorithms to be implemented. Departing from this approach, a recent innovation with respect to languages for constraint programming is in the design of the OPL Optimization Programming Language [Van Hentenryck 1999], where the language was designed with the purpose of making it easy to solve optimization problems by supporting constraint programming and mathematical programming techniques. Here, the completeness of the language for computer programming as well as the ability to program constraint solving algorithms was not deemed to be as important. Instead, the language is designed to support the declarative representation of optimization problems, and includes the facilities to use an underlying constraint programming engine, with the ability to describe via computer programming techniques a search strategy to find solutions to problems. The OPL language is not a complete computer programming language, 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 presented here are related to those in the book describing OPL by Van Hentenryck [1999].

The design of OPL was motivated by the increased interest in mathematical programming modeling languages such as AMPL [Fourer, et., al., 1993] and GAMS [Bisschop and Meeraus, 1982] in combination with the recent success of using constraint programming in the context of solving combinatorial optimization problems. These NP-hard problems include feasibility problems as well as optimization problems. With regards to optimization, constraint programming is often applied in situations where a quick feasible solution to a problem is required, as opposed to finding 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 provides an alternative technique to integer programming for solving a variety of combinatorial optimization problems.

The books by Van Hentenryck [1989], Marriott and Stuckey [1999] and Hooker [2000] provide readers with descriptions of some of the underlying theory of constraint programming. The book about OPL by 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.

3 Constraint Programming Formulations

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

3.1 Constraint Satisfaction Problems

We first formally define a constraint satisfaction problem, using some of the terminology of mathematical programming. Given a set of n decision variables $x_1,x_2,\ldots,x_n$, 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, John, Jim, Jack}. There is no restriction on the type of each decision variable, and hence decision variables 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, i.e., 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:

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

$$f_k(x_1,x_2,\ldots,x_n)=1, \quad 1 \le k \le m$$
$$x_j \in D_j \qquad 1 \le j \le n$$

Note that this problem is only a feasibility problem, and that no objective function is defined. Nevertheless, CSPs are an important class of combinatorial optimization problems. It is important to note here that the functions f_k do not necessarily have closed mathematical forms (e.g., functional representations), and can simply be defined 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.

3.2 Optimization problems

In the previous section, we defined a constraint satisfaction problem as a feasibility problem. With regards to optimization, constraint programming systems also allow an objective function to be specified. 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

$$\begin{array}{ll} \text{Minimize} & g(x_1,x_2,\ldots,x_n) \\ \text{Subject to} & f_k(x_1,x_2,\ldots,x_n)=1, \qquad 1 \le k \le m \\ & x_j \in D_j \qquad\qquad 1 \le j \le n \end{array}$$

3.3 Simple examples

We now present some simple examples of constraint satisfaction and optimization problems using constraint programming techniques. In order to do so, 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 desire to indicate the relationships between constraint programming and mathematical programming, we will use the OPL Optimization Programming Language for these representations. A discussion of the issues with regards to programming a search strategy is postponed until Section 4.3. For the examples in this section, we note that the OPL language includes a default search strategy for finding a solution to the problem, which removes the need for always describing such a strategy. Section 4.3.1 provides an example where a search strategy is required in order to be able to solve the problem. In our presentations of OPL, line numbers are included in the left column that are not part of the language, but rather make it easier to discuss the specific problems.

3.3.1 Map Coloring

Our first example comes from map coloring, which is known to be 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. It is desired to determine an assignment of the colors to the countries so that no two adjacent countries have the same color. An OPL statement for this problem is shown in Figure 1. Lines 1 and 2 respectively specify the set of countries and colors and that these sets are 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. Hence, the data is specified as a list of pairs of adjacent countries.

Line 8 contains the decision variables for the problem. Here, there is 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`. Note that the domain of each variable 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 10 will be stated for each member of the set

neighbors, i.e., for each pair of countries that are adjacent. The set of 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.

```
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    };
```

Figure 1: The Map Coloring Problem

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, these relations are usually expressed by introducing additional binary variables and additional constraints into a mixed integer programming representation.

3.3.2 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, e.g., [Gusfield and Irving 1989]. Given a set of *n* women and an equal number of men, we assume that each of the women have 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 have 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) 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) 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 is solved annually each year by the National Resident Matching Program.

The conditions for stability are easily stated as a constraint satisfaction problem in OPL and are shown in Figure 2. Lines 1 and 2 respectively specify the set of women and men. The elements of these sets can be used to index into arrays, and hence the two-dimensional array rankWomen specifies the preference rankings by the women of the men, while the two-dimensional array rankMen specifies the preference rankings by the men of the women. The decision variables for this problem, specified on 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. Note that 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 desire to find a solution to a constraint satisfaction problem. Again, the forall constructor is used to indicate a group of constraints. On Lines 8 and 9, a constraint is stated that specifies that the husband of the wife of man m must be man m. A similar constraint is stated 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.

```
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   };
```

Figure 2: The Stable Marriage Problem

3.3.3 Sequencing

Consider a set of blocks in a plant that need to be painted. Each block has a designated color and it is desired to sequence the blocks in such a way so as to minimize the cost of the number of times that the paint color is changed 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. Figure 3 provides an OPL model for this problem. 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) that each block can be placed in. This type can then act as a set of integers, just like the type `Colors` represents the set of colors. Line 4 declares the array `whenblock`, which corresponds to the values w_i. It is assumed 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 mid-sequence. Lines 10 and 11 determine for each possible color, the set of blocks that need 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.

```
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  };
```

Figure 3: A Sequencing Problem

Lines 24 and 25 illustrate an example of a *global constraint.* The constraint `alldifferent(position)` indicates that each member of the `position` array should have a different value. In essense, 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 easily seen that blocks of the same color should be ordered in the same order as 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]`.

4 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. As mentioned earlier, traditional constraint programming systems require that the user 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, there is a default search strategy that is used if the user does not provide a search strategy. Programming a search strategy is often needed in order to effectively apply constraint programming to solve a problem. In this section, we will discuss first the fundamental algorithm underlying a constraint programming system, and then indicate the methodologies used to program search strategies.

4.1 Constraint Propagation and Domain Reduction

In Section 3, a constraint was defined 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 *propagated* to any constraint that interacts with that variable. For each constraint, a *domain reduction algorithm* is then programmed that 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 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 in Brearley, et. al. [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.

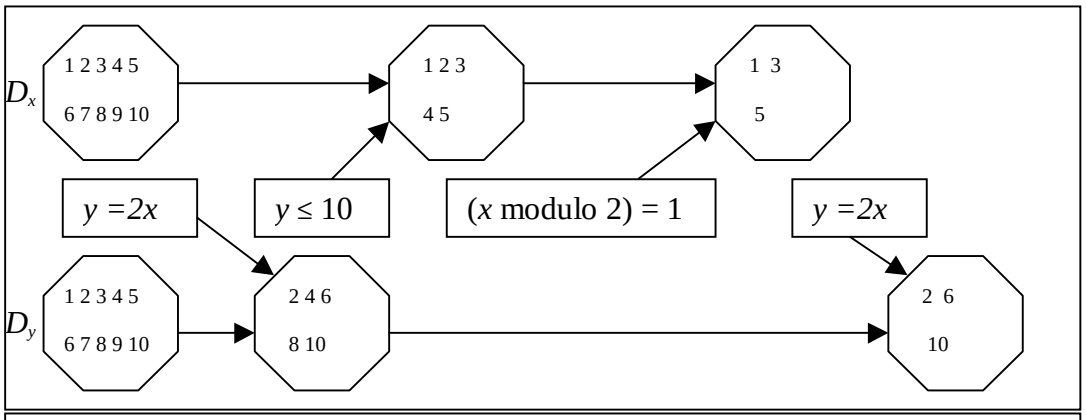

Figure 4: Illustration of Constraint Propagation and Domain Reduction

The methodology is best demonstrated by an example, illustrated in Figure 4. 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 then follows that $x \leq 5$, and hence the domain of x can be changed to $D_x=\{1,2,3,4,5\}$. Suppose that now 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 then remove the values of 4 and 8 from the domain of y and obtain $D_y=\{2,6,10\}$.

Some constraint programming systems (e.g., ILOG Solver, Oz, ECLiPSe) allow the programmer to take advantage of existing propagators for built-in constraints that cause domain reductions, and to build one's own propagation and domain reduction schemes for user-defined constraints. However, many constraint programming systems (for example, OPL, ILOG Solver, CHIP) are now strong enough that large libraries of predefined constraints, with associated propagation and domain reduction algorithms, are provided as part of the system, 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, a value $d \in D_j$ is *consistent* with f_k if there is at least one assignment of the variables 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 CSP's 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.

A number of algorithms have been developed to efficiently propagate constraints and reduce domains so as to create systems that are arc consistent. One of the algorithms is called AC-5, developed by Van Hentenryck, et. al. [1992]. This latter article is significant in the constraint programming context 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.

4.2 Constraint Programming Algorithms for Optimization

A weakness of a constraint programming approach when applied to a problem with an objective function is that there is not necessarily a lower bound present when minimizing the objective function. 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, where a lower bound always exists due to 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 used 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 constraint that is added 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 until 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, an initial feasible point is found, which determines an upper bound U on the objective function. A dichotomic search procedure is essentially a binary search on the objective function. The midpoint $M=(U+L)/2$ of the two bounds is computed, and a CSP is solved by taking the original constraints and adding the constraint $g(x_1,x_2,\ldots,x_n)<M$. If a new feasible point is found, then the upper bound is updated, and the search continues in the same way with a new midpoint M. If the system is found to be infeasible, then the lower bound is updated, 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 might 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 is stressing the search for feasible solutions, whereas branch-and-bound procedures usually emphasize the improvement of the lower bound.

4.3 Programming Search

Given a CSP, the constraint propagation/domain reduction algorithm can be applied to reduce the domains of the variables so as to arrive at an arc consistent system. However, while this may determine if the CSP is infeasible, it does not necessarily find solutions of a CSP. To do this, one must program a search strategy (or use a default search strategy, if one is provided by the constraint programming system). Traditionally, the search facilities provided by a constraint programming system 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 at a node low in the tree a solution is found, or until the entire tree is explored, in which case the CSP is found to be infeasible. The search strategy is enumerative with constraint propagation and domain reduction employed at each node to help prune the search space.

```
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    };
```

Figure 5: A simple CSP

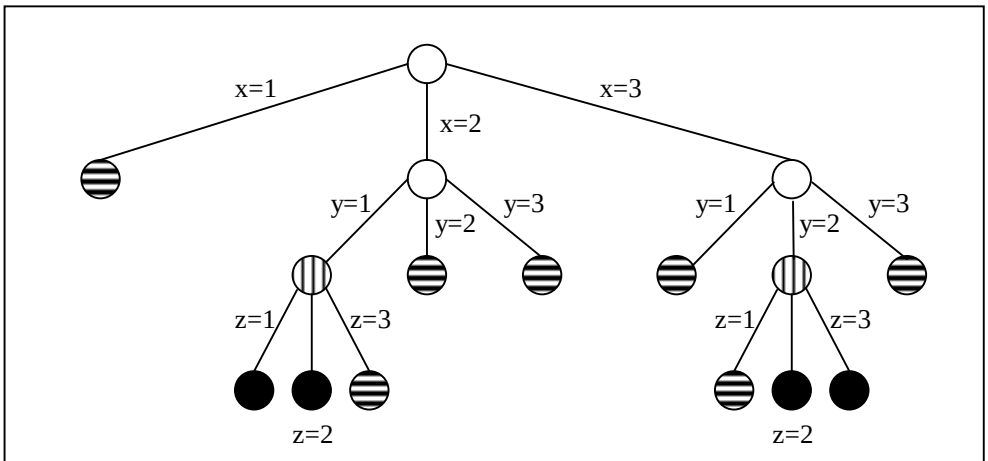

Figure 6: A search tree

A simple example will illustrate this idea. Consider the OPL example shown in Figure 5, which shows a simple CSP on 3 variables, each with the same domain. 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 illustrated in Figure 6. 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 then immediately begin

A recent innovation in constraint programming systems is found in ILOG Solver 4.4 [ILOG 1999] and ILOG OPL Studio 2.1, where the idea of allowing the programmer to use other strategies beyond depth first search is provided. Depth first search has traditionally been used because in the context of viewing constraint programming as a particular computer programming methodology, the issues regarding memory management are dramatically simplified. ILOG Solver 4.4 and OPL Studio 2.1 (which uses ILOG's Solver technology) allows 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 some global limit is reached (e.g., time, node count, etc.) With these basic constructs in place, it is then possible to easily program any search strategy that systematically searches the entire search space by choosing nodes to explore (i.e., programming node evaluators), dividing the search space at nodes (i.e., programming goals and creating choice points), and picking the choice to evaluate next within a specific node (i.e., programming search selectors). Constraint programming systems provide a framework for describing enumeration strategies for solving search problems in combinatorial optimization. The connection between search strategies in constraint programming and branch-and-bound procedures for mixed integer programming is discussed in Section 5.1.

4.3.1 An Example of Search: Graceful Graphs

Given a graph $G=(V,E)$ with $n=|V|$ nodes and $m=|E|$ edges, the graph is said to be *graceful* if there are unique node labels $f:V\rightarrow\{0,1,2,...,m\}$ and unique edge labels $g:E\rightarrow\{1,2,...,m\}$ such that $g(i,j)=|f(i)-f(j)|$ for each edge $e\in E$ with $e=(i,j)$. Graceful graphs have applications in radio astronomy and cryptography. An example of a graph and a graceful labeling for this graph is given in Figure 7. The numbers in an italic font are the labels for the nodes, while the other numbers are the labels for the edges.

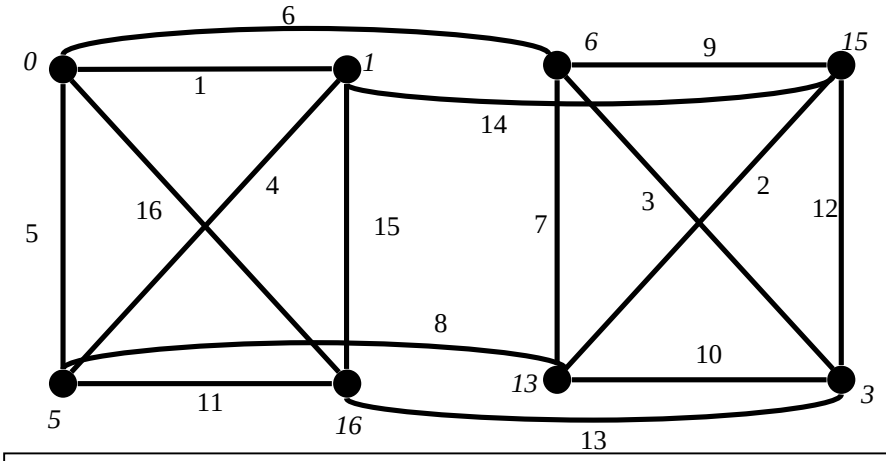

Figure 7: A graph and a graceful labeling

Figure 8 provides an OPL solution to the problem of finding a graceful labeling of a graph. 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 illustrated in Figure 7. 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.

```
1   int numnodes = ...;
2   range Nodes 1..numnodes;
3   struct Edge {
4      Nodes  i;
5      Nodes  j;
6   };
7   setof(Edge) edges = ...;
8   int numedges = card(edges);
9   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  };
```

Figure 8: Graceful Graphs

4.3.2 An Example of Search: 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. Figure 9 illustrates an OPL model for this problem that uses constraint programming constructs, and a search strategy for solving the problem. 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 are 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. Note that 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, that 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. In Section 5.2, we will see an integer programming formulation of the same problem.

```
1   int   fixed = ...;
2   int   nbStores = ...;
3   enum  Warehouses ...;
4   range Stores 1..nbStores;
5   int   capacity[Warehouses] = ...;
6   int   supplyCost[Stores,Warehouses] = ...;
7   int  maxCost = max(s in Stores, w in Warehouses) supplyCost[s,w];
8   Warehouses wlist[w in Warehouses] = w;
9
10  var int open[Warehouses] in 0..1;
11  var Warehouses supplier[Stores];
12  var int cost[Stores] in 0..maxCost;
13
14  minimize
15    sum(s in Stores) cost[s] + sum(w in Warehouses) fixed * open[w]
16  subject to {
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
24  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  };
```

Figure 9: Constraint Programming Formulation of Warehouse Location Problem

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, try different warehouses that it can be assigned to, 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. By considering a particular store `s`, it is easy to 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]`, by considering each possible warehouse `w`. Hence, if there were 5 warehouses, the initial domain of `cost[s]` would have 5 values. As the search progresses for 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 of 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, i.e., 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 then a different warehouse will be tried 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 respectively determined by the constraints in lines 18 and 20, so no specific search strategy for those variables is required.

The search strategy described above uses knowledge about the problem to guide the search. First, it chooses to assign stores to warehouses, and then 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. Without this search strategy on one particular instance, 1894 choice points are required to find a solution. With this search strategy, only 147 choice points are needed.

5 Connections to Integer Programming

Our description of constraint programming has emphasized how it 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. This section discusses some of these connections.

5.1 Constraint Programming and Branch and Bound

For those familiar with integer programming, the concept of search strategies as described in Section 4.3 should seem familiar. In fact, branch and bound, which is an enumerative search strategy, has been used to solve integer programs since the mid 1960's. Lawler and Wood [1966] give an early survey, while the classic text by Garfinkel and Nemhauser [1972] describes branch and bound in the context of an enumerative procedure. A more recent discussion is given by Nemhauser and Wolsey [1988]. In systems that have been developed for integer programming, users are often given the option of selecting a *variable selection strategy* and a *node selection strategy*. These are clearly equivalent to the descriptions of *search selectors* and *node evaluators* described above.

There are two fundamental ways that a constraint programming framework extends the basic branch-and-bound procedures that are implemented in typical mixed integer programming solvers. First, in most implementations of branch and bound procedures for mixed integer programming, two branches are created at each node after a variable x with a fractional value v has been chosen to branch on. The search space is then divided in two parts, by creating a choice point based on the two choices of $(x \le \lfloor v \rfloor)$ and $(x \ge \lceil v \rceil)$. 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, a choice point could be created 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 a strategy is programmed

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 programming the search strategies. Combining linear programming and constraint programming has appeared in Prolog III [Colmerauer 1990], CLP(R) [Jaffar & Lassez 1987], CHIP [Dincbas, Van Hentenryck, et. al. 1988], ILOG Solver and ILOG Planner [ILOG, 1999] and ECLiPSe [Wallace, et. al. 1997].

```
1   int   fixed = ...;
2   int   nbStores = ...;
3   enum  Warehouses ...;
4   range Stores 1..nbStores;
5   int   capacity[Warehouses] = ...;
6   int   supplyCost[Stores,Warehouses] = ...;
7
8   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) supplyCost[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  };
```

Figure 10: Integer Programming Formulation of Warehouse Location Problem

It should be noted that a number of branch-and-bound implementations for mixed integer programming have allowed users to program custom search strategies, including MINTO [Nemhauser, Savelsbergh, 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 in 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 as well as 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.

5.2 Contrasting Formulations

In Section 4.3.2, we illustrated an example of an optimization problem using constraint programming techniques. It is worth contrasting this formulation with a pure integer programming formulation, which is shown in Figure 10. 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. Note that 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 only be assigned 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.

It is interesting to contrast the two formulations. 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 are written to enforce this restriction. In the integer programming formulation, stores are assigned to open warehouses by using 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 constraint is written as a set of linear constraints in the integer programming formulation.

A natural question to ask is whether one formulation is better than the other. 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 is sometimes dependent 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. More research is needed to better understand how to make this kind of comparison. In the next section, we illustrate a hybrid approach that combines both formulations.

6 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. In Section 5.1, 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 where the two approaches can cooperate. The first, described in Section 6.1, is in stating formulations that contain a mixture of linear and "not-linear" constraints. The second, described in Section 6.2, is where a problem is decomposed so that constraint programming is used to solve one part of the problem, and mathematical programming is used to solve a second part.

6.1 A Hybrid Formulation of the Warehouse Location Problem

Another way of solving the warehouse location problem is to combine the two formulations. The OPL model shown in Figure 11 omits the data section and the search strategy (which is the same as in Section 4.3.2), and just shows the decision variables and the constraints. 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 (Note that dichotomic search is not used 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 prune the search space, and reduce the overall solution time. For example, a simple 5 warehouse, 5 store problem is solved using the formulation in Figure 9 using 147 choice points, while the same problem is solved using the formulation in Figure 11 using 53 choice points. For this particular data set, all 3 formulations are solved in a fraction of a second, making the comparison of formulations difficult.

With the advance in the available technology, exploration of these kinds of mixed formulations is now possible, and might prove to be an effective solution technique for hard combinatorial optimization problems that benefit from multiple representations. Jain and Grossman [2000] have investigated this for some machine scheduling applications.


```
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  };
```

Figure 11: Hybrid Formulation of Warehouse Location Problem

6.2 Column Generation and Crew Scheduling

One of the interesting applications of constraint programming is in the context of applying column generation approaches to solve different kinds of combinatorial optimization problems. The classic example is the cutting stock problem, where knapsack problems are used to generate the possible patterns that can be used and linear programs are then solved to determine how many cuts of each pattern should be used. The dual solutions to each linear program change the cost structure of the knapsack problem, which is solved to generate a new column of the linear program.

Another example can be found in crew scheduling applications. For a number of years, crew scheduling problems have been solved by writing a computer program to generate the potential pairings of crews to flights. The program has to generate pairings, corresponding to a sequence of flights for a single crew, that are low in cost and satisfy the complex duty rules that are written in volumes of regulations specified by the carrier. The typical approach that has been used in the past and is still in use today is a "Generate and Test" approach, where a complete pairing is generated, and the feasibility of this pairing is then tested against all of the rules programmed into the system. After a suitable number of pairings have been generated, a set partitioning problem is then solved by letting each pairing correspond to one column of the set partitioning problem.

Here, we describe a methodology where constraint programming can be applied to do the pairing generation. Given a set of flights and a flight schedule, variables can be declared that correspond to the sequence of flights covered by one pairing. Constraints can then be stated that dictate the rules about sequences of flights, as well as other rules related to how many hours can be in a total sequence of flights, the required time between flights, etc. A search procedure can then be carried out to generate potential flight sequences. As the search proceeds, the underlying constraint propagation and domain reduction algorithms are used to 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 a suitable set of pairings has been generated, these can then be used 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/`.

We believe that many operations research applications involve various kinds of enumeration strategies that are usually embedded in complex computer programs, and that constraint programming provides an attractive alternative for implementing these kinds of enumeration schemes. We prefer to call this *constrained enumeration.*

7 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, where 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 framework to directly represent these problems.

```
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  };
```

Figure 12: Jobshop scheduling problem

The jobshop problem is given in OPL in Figure 12. The first 9 lines define the input data of the problem, consisting of the number of machines, number of jobs, and number of tasks. The OPL keyword `range` is used 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 is stating 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 scheduling specific constraint propagation and domain reduction algorithms helps to immediately prune the search space and determine bounds on the start and end times of activities. Secondly, in practice it is not necessarily required that a provably optimal solution be found to such problems, but that a good feasible solution is quickly found. 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.

8 Summary

Our experience indicates that constraint programming is better than integer programming on applications relating to sequencing and scheduling, as well as for problems where there is much symmetry in an integer programming formulation. 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 where the linear programming relaxations provide strong bounds for the objective function. Hybrid techniques are relatively new, although some recent work by Jain and Grossman [2000] describes some machine scheduling applications where they have proven to be effective.

As evidenced by recent sessions at INFORMS meetings, there has been growing interest in how constraint programming technologies and mathematical programming approaches can complement each other. Because of the introduction of languages like OPL and systems like ECLiPSe, it is now possible to easily explore alternative and hybrid approaches to solving difficult problems. In the future, we think that this development will continue, and that additional difficult industrial problems will be solved using combinations of both techniques. It is our hope that constraint programming will become as familiar to operations research professionals as linear programming is today.

9 Bibliography

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* 8, pp. 54-83.

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

Colmerauer, Alain, 1990, "An introduction to PROLOG III," *Communications of the ACM* 33(7), pp. 70-90.

Dantzig, George B. 1948, "Programming in a Linear Structure," Comptroller, USAF, Washington, D.C., February, 1948.

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, 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.

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)*, volume 1, pp. 607-613.

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

Jain, Vipul and Grossman, Ignacio, 2000, "Algorithms for Hybrid MILP/CP Models for a Class of Optimization Problems," Preprint, Department of Chemical Engineering, Carnegie Mellon University.

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

Knuth, Donald E. 1968, *Fundamental Algorithms, The Art of Computer Programming*, Volume 1, Second Edition, Addison-Wesley, Reading, Massachusetts.

Lauriere, Jean-Louis, 1978, "A Language and a Program for Stating and Solving Combinatorial Problems," *Artificial Intelligence* 10(1), pp. 29-127.

Lawler, Eugene L., and Wood, D.E. 1966, "Branch-and-Bound Methods: A Survey," *Operations Research* 14, pp. 699-719.

Mackworth, Alan K. 1977, "Consistency in networks of relations," *Artificial Intelligence* 8, 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)*, volume 2, pp. 1382-1387.

Nemhauser, George W. and Wolsey, Laurence A., 1972, *Integer Programming*, Wiley, New York.

Nemhauser, George W., Savelsbergh, Martin W.P., and Sigismondi, Gabriele C., 1994, "MINTO, a Mixed INTeger Optimizer," *Operations Research Letters* 15, 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 First 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", in *Logic Programming: Proceedings of the 1995 International Symposium*, John Lloyd, editor, MIT Press, Cambridge, 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, 24-25 June 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* 57(2), pp. 291.
Wallace, Mark, Novello, Stefano, and Schimpf, Joachim, 1997, "ECLiPSe : A Platform for Constraint Logic Programming", *ICL Systems Journal*, 12(1), pp. 159-200.
Walsh, Toby, 1997, "Depth-bounded discrepancy search," *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, volume 2, pp. 1388-1393.
Williams, H.P., 1999, *Model Building in Mathematical Programming, 4th Edition*, John Wiley & Sons, New York.

