
# SAS/OR® 14.3 User's Guide
## Mathematical Programming
# The Constraint Programming Solver

This document is an individual chapter from *SAS/OR® 14.3 User's Guide: Mathematical Programming*.

The correct bibliographic citation for this manual is as follows: SAS Institute Inc. 2017. *SAS/OR® 14.3 User's Guide: Mathematical Programming*. Cary, NC: SAS Institute Inc.

**SAS/OR® 14.3 User's Guide: Mathematical Programming**



SAS Institute Inc., SAS Campus Drive, Cary, NC 27513-2414

September 2017

# Chapter 6
# The Constraint Programming Solver

**Contents**

# Overview: CLP Solver

You can use the constraint logic programming (CLP) solver in the OPTMODEL procedure to address finite-domain constraint satisfaction problems (CSPs) that have linear, logical, and global constraints. In addition to providing an expressive syntax for representing CSPs, the CLP solver features powerful built-in consistency routines and constraint propagation algorithms, a choice of nondeterministic search strategies, and controls for guiding the search mechanism. These features enable you to solve a diverse array of combinatorial problems.

Many important problems in areas such as artificial intelligence (AI) and operations research (OR) can be formulated as constraint satisfaction problems. A CSP is defined by a finite set of variables that take values from finite domains and by a finite set of constraints that restrict the values that the variables can simultaneously take.

A solution to a CSP is an assignment of values to the variables in order to satisfy all the constraints. The problem amounts to finding one or more solutions, or possibly determining that a solution does not exist.

A constraint satisfaction problem (CSP) can be defined as a triplet $\langle X, D, C \rangle$:

- $X = \{x_1, \ldots, x_n\}$ is a finite set of *variables*.
- $D = \{D_1, \ldots, D_n\}$ is a finite set of *domains*, where $D_i$ is a finite set of possible values that the variable $x_i$ can take. $D_i$ is known as the *domain* of variable $x_i$.
- $C = \{c_1, \ldots, c_m\}$ is a finite set of *constraints* that restrict the values that the variables can simultaneously take.

The domains do not need to represent consecutive integers. For example, the domain of a variable could be the set of all even numbers in the interval [0, 100]. In PROC OPTMODEL, variables are always numeric. Therefore, if your problem contains nonnumeric domains (such as colors), you must map the values in the domain to integers. For more information, see "Example 6.5: Car Painting Problem" on page 229.

You can use the CLP solver to find one or more (and in some instances, all) solutions to a CSP that has linear, logical, and global constraints. The numeric components of all variable domains are required to be integers.

# Getting Started: CLP Solver

The following examples illustrate the use of the CLP solver in formulating and solving two well-known logical puzzles in constraint programming.

## Send More Money

The Send More Money problem consists of finding unique digits for the letters D, E, M, N, O, R, S, and Y such that S and M are different from zero (no leading zeros) and the following equation is satisfied:

S E N D

\+ M O R E

---

M O N E Y

You can use the CLP solver to formulate this problem as a CSP by using an integer variable to represent each letter in the expression. The comments before each statement in the following code introduce variables, domains, and constraints:

```
/*  Send More Money  */

proc optmodel;
   /* Declare all variables as integer. */
   var S integer, E integer, N integer, D integer, M integer, O integer,
       R integer, Y integer;
   /* Set all domains to between 0 and 9. Domains are unbounded by default.
      Always declare domains to be as tight as possible. */
   for {j in 1.._NVAR_} do;
      _VAR_[j].lb = 0;
      _VAR_[j].ub = 9;
   end;
   /* Describe the arithmetic constraint.*/
   con Arithmetic:           1000*S + 100*E + 10*N + D
                  +          1000*M + 100*O + 10*R + E
                  = 10000*M + 1000*O + 100*N + 10*E + Y;
   /* Forbid leading letters from taking the value zero.
      Constraint names are optional. */
   con S ne 0;
   con M ne 0;
   /* Require all variables to take distinct values. */
   con alldiff(S E N D M O R Y);

   solve;
   print S E N D M O R Y;
quit;
```

The domain of each variable is the set of digits 0 through 9. The VAR statement identifies the variables in the problem. The Arithmetic constraint defines the linear constraint SEND + MORE = MONEY and the restrictions that S and M cannot take the value 0. (Alternatively, you can simply specify the domain for S and M as $\geq 1 \leq 9$ in the VAR statement.) Finally, the ALLDIFF predicate enforces the condition that the assignment of digits should be unique.

The PRINT statement produces the solution output as shown in Figure 6.1.

**Figure 6.1** Solution to SEND + MORE = MONEY

**The OPTMODEL Procedure**

| Problem Summary | |
|---|---|
| **Objective Sense** | Minimization |
| **Objective Function** | (0) |
| **Objective Type** | Constant |
| | |
| **Number of Variables** | 8 |
| **Bounded Above** | 0 |
| **Bounded Below** | 0 |
| **Bounded Below and Above** | 8 |
| **Free** | 0 |
| **Fixed** | 0 |
| **Binary** | 0 |
| **Integer** | 8 |
| | |
| **Number of Constraints** | 4 |
| **Linear LE (<=)** | 0 |
| **Linear EQ (=)** | 1 |
| **Linear GE (>=)** | 0 |
| **Linear LT (<)** | 0 |
| **Linear NE (~=)** | 2 |
| **Linear GT (>)** | 0 |
| **Linear Range** | 0 |
| **Alldiff** | 1 |
| **Element** | 0 |
| **GCC** | 0 |
| **Lexico (<=)** | 0 |
| **Lexico (<)** | 0 |
| **Pack** | 0 |
| **Reify** | 0 |
| | |
| **Constraint Coefficients** | 10 |

| Solution Summary | |
|---|---|
| **Solver** | CLP |
| **Variable Selection** | MINR |
| **Objective Function** | (0) |
| **Solution Status** | Solution Limit Reached |
| **Objective Value** | 0 |
| | |
| **Solutions Found** | 1 |
| **Presolve Time** | 0.00 |
| **Solution Time** | 0.00 |

| S | E | N | D | M | O | R | Y |
|---|---|---|---|---|---|---|---|
| 9 | 5 | 6 | 7 | 1 | 0 | 8 | 2 |

The CLP solver determines the following unique solution to this problem:

$$\begin{array}{r} 9\,5\,6\,7 \\ +\,1\,0\,8\,5 \\ \hline 1\,0\,6\,5\,2 \end{array}$$

## Eight Queens

The Eight Queens problem is a special instance of the $N$-Queens problem, where the objective is to position $N$ queens on an $N \times N$ chessboard such that no two queens can attack each other. The CLP solver provides an expressive constraint for variable arrays that can be used for solving this problem very efficiently.

You can model this problem by using a variable array $\boldsymbol{a}$ of dimension $N$, where $a_i$ is the row number of the queen in column $i$. Because no two queens can be in the same row, it follows that all the $a_i$'s must be pairwise distinct.

In order to ensure that no two queens can be on the same diagonal, the following two expressions should be true for all $i$ and $j$:

$$a_j - a_i <> j - i$$

$$a_j - a_i <> i - j$$

These expressions can be reformulated as follows:

$$a_i - i <> a_j - j$$

$$a_i + i <> a_j + j$$

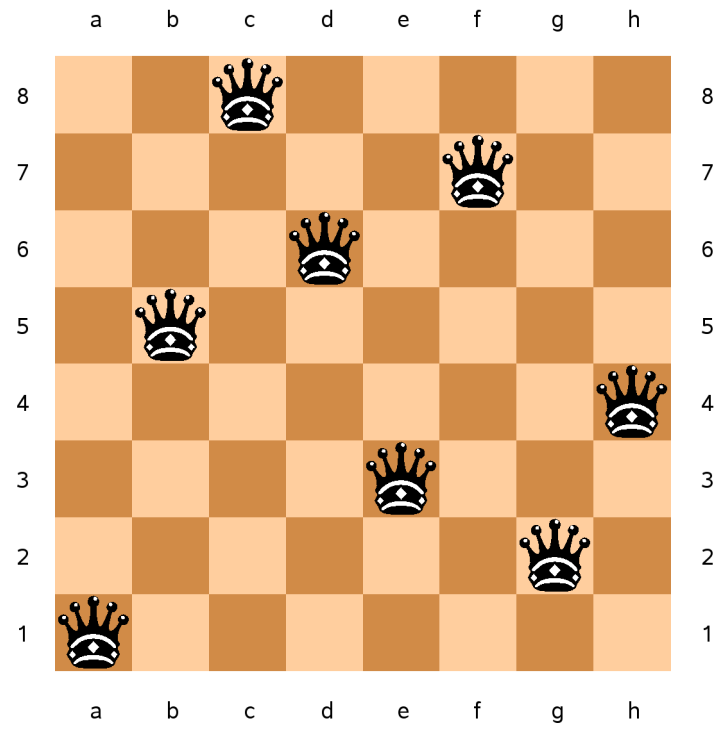
Hence, the $(a_i + i)$'s are pairwise distinct, and the $(a_i - i)$'s are pairwise distinct. One possible formulation is as follows:

```
proc optmodel;
   num n init 8;
   var A {1..n}        >= 1     <= n     integer;
   /* Define artificial offset variables. */
   var B {1..n, -1..1} >= 1 - n <= n + n integer;
   con Bdef {i in 1..n, k in -1..1}:
      B[i,k] = A[i] + k * i;

   con OffsetsMustBeAlldifferent {k in -1..1}:
      alldiff({i in 1..n} B[i,k]);
   solve with CLP / varselect=fifo;
   /* Replicate typical PROC CLP output from an OPTMODEL array */
   create data out from {i in 1..n}<col('A'||i)=A[i]>;
quit;
```

The VARSELECT= option specifies the variable selection strategy to be first-in, first-out—the order in which the CLP solver encounters the variables.

The corresponding solution to the Eight Queens problem is displayed in Figure 6.2.

**Figure 6.2** A Solution to the Eight Queens Problem



# Syntax: CLP Solver

**SOLVE WITH CLP** /
< **FINDALLSOLNS** >
< **MAXSOLNS=***number* >
< **MAXTIME=***number* >
< **NOPREPROCESS** >
< **OBJTOL=***number* >
< **PREPROCESS** >
< **SHOWPROGRESS** >
< **TIMETYPE=***number* | *string* >
< **VARASSIGN=***string* >
< **VARSELECT=***string* >
;

The section "Functional Summary" on page 197 provides a quick reference for each option. Each option is then described in more detail in its own section, in alphabetical order.

## Functional Summary

Table 6.1 summarizes the options available in the SOLVE WITH CLP statement.

**Table 6.1** Functional Summary of SOLVE WITH CLP Options

| Description | Option |
|---|---|
| **General Options** | |
| Finds all possible solutions | FINDALLSOLNS |
| Specifies the number of solution attempts | MAXSOLNS= |
| Specifies the maximum time to spend calculating results | MAXTIME= |
| Suppresses preprocessing | NOPREPROCESS |
| Specifies the tolerance of the objective value | OBJTOL= |
| Permits preprocessing | PREPROCESS |
| Indicates progress in the log | SHOWPROGRESS |
| Specifies whether time units are CPU time or real time | TIMETYPE= |
| Specifies the variable assignment strategy | VARASSIGN= |
| Specifies the variable selection strategy | VARSELECT= |

## SOLVE WITH CLP Statement

**SOLVE WITH CLP** / < *options* > ;

The SOLVE WITH CLP statement invokes the CLP solver. You can specify the following *options* to define various processing and diagnostic controls and to tune the algorithm to run.

## General Options

You can specify the following general options.

**FINDALLSOLNS**

**ALLSOLNS**

**FINDALL**

attempts to find all possible solutions to the CSP.

**MAXSOLNS=***number*

specifies the number of solution attempts to be generated for the CSP. By default, MAXSOLNS=1.

**MAXTIME=***number*

specifies the maximum time to spend calculating results. The type of time (either CPU time or real time) is determined by the value of the TIMETYPE= option. The value of *number* can be any positive number; the default value is the positive number that has the largest absolute value that can be represented in your operating environment.

**NOPREPROCESS**

suppresses any preprocessing that would usually be performed for the problem.

**OBJTOL=***number*

specifies the tolerance of the objective value. By default, OBJTOL=1E–6.

**PREPROCESS**

permits any preprocessing that would usually be performed for the problem.

**SHOWPROGRESS**

prints a message to the log whenever a solution is found.

**TIMETYPE=***number* | *string*

specifies whether to use CPU time or real time for the MAXTIME= option. Table 6.2 describes the valid values of the TIMETYPE= option.

**Table 6.2** Values for TIMETYPE= Option

| *number* | *string* | **Description** |
|---|---|---|
| 0 | CPU | Specifies units of CPU time |
| 1 | REAL | Specifies units of real time |

By default, TIMETYPE=REAL.

**VARASSIGN=***string*

specifies the variable assignment strategy. You can specify two value selection strategies:

- MAX, which selects the maximum value from the domain of the selected variable
- MIN, which selects the minimum value from the domain of the selected variable

By default, VARASSIGN=MIN.

**VARSELECT=***string*

specifies the variable selection strategy. The strategy could be static, dynamic, or conflict-directed. Typically, static strategies exploit information about the initial state of the search, whereas dynamic strategies exploit information about the current state of the search process. Conflict-directed strategies exploit information from previous states of the search process as well as the current state (Boussemart et al. 2004). Table 6.3 describes the valid values of the VARSELECT= option.

**Table 6.3** Values for VARSELECT= Option

| *string* | **Description** |
|---|---|
| **Static strategies** | |
| FIFO | Uses the first-in, first-out ordering of the variables as encountered by the procedure after adjusting for the values in the .priority suffix |
| MAXCS | Selects the variable that has the maximum number of constraints |
| **Dynamic strategies** | |
| DOMDDEG | Selects the variable that has the smallest ratio of domain size to dynamic degree |
| DOMDEG | Selects the variable that has the smallest ratio of domain size to degree |
| MAXC | Selects the variable that has the largest number of active constraints |
| MINR | Selects the variable that has the smallest range (that is, the minimum value of the upper bound minus the lower bound) |
| MINRMAXC | Selects the variable that has the smallest range, breaking ties by selecting one that has the largest number of active constraints |
| **Conflict-directed strategies** | |
| DOMWDEG | Selects the variable that has the smallest ratio of domain size to weighted degree |
| WDEG | Selects the variable that has the largest weighted degree |

The dynamic strategies embody the "fail-first principle" (FFP) of Haralick and Elliott (1980), which suggests, "To succeed, try first where you are most likely to fail." By default, VARSELECT=MINR.

You can change the order in which the CLP solver selects variables after you have declared them by setting the .priority suffix for your variables. Variables with higher priority are selected first.

PROC OPTMODEL usually honors the variable declaration order when it invokes the CLP solver. However, when variables are indexed over dynamically computed sets, the variable order might be undefined. The value of .priority always overrides declaration order. To ensure that the CLP solver considers variables in the exact order that you want, set the .priority suffix for all variables. You can also experiment with the priorities by changing the suffixes without having to reorder the variable declarations in your code.

For more information, see the section "Integer Variable Suffixes" on page 136 in Chapter 5, "The OPTMODEL Procedure."

## Predicates

A predicate asserts a fact about its subject. You use a predicate as the first identifier in a constraint declaration to define what must hold true in any feasible solution. The following is an example of a constraint that uses the ELEMENT predicate:

```
var X integer, Y integer;
con ElementConstraintExample:   element(X, 1 2 3 4, Y);
```

The following predicates are available in PROC OPTMODEL:

**ALLDIFF(***variable-list***)**
**ELEMENT(***scalar-variable***,***data-list***,***scalar-variable***)**
**GCC(***variable-list***,***set-of-numeric-triplets***)**
**LEXICO(***variable-list order-type variable-list***)**
**PACK(***variable-list***,***data-list***,***variable-list***)**
**REIFY(***scalar-variable***,** *linear-constraint***)**

### Common Syntax Components

The following syntax components are used in multiple predicates. They depend on the definition of an *identifier-expression*. For more information, see the section "Identifier Expressions" on page 98 in Chapter 5, "The OPTMODEL Procedure." For information about other syntactic components, see the section that describes the corresponding predicate.

*data-list*

is a space-separated list of items, each of which can be prefixed by an indexing set and conforms to one of the following syntaxes:

- a number
- an *identifier-expression*, but excluding suffixes
- the name of a numeric array
- ( *expression* ), which must evaluate to a number

For example, the first three constraints in the following statements refer to valid *data-lists*, whereas the last four do not. Each incorrect constraint is preceded by a comment that explains why the constraint is incorrect.

```
var X integer, Y integer, Z integer;
num n;
con CorrectDataList1:   element(X, 1 2 3 4, Y);
con CorrectDataList2:   element(X, 1 2 3 4 n, Y);
con CorrectDataList3:   element(X, {i in 1..4} i n, Y);
/* The parentheses imply a scalar expression */
con IncorrectDataList1: element(X, ({i in 1..4} i) n, Y);
/* [1 2 3 4] is an array initializer, not an array name */
con IncorrectDataList2: element(X, [1 2 3 4], Y);
/* Z is a variable */
con IncorrectDataList3: element(X, 1 2 3 4 Z, Y);
/* Impossible to distinguish Z.sol from [ Z . sol ] */
```

```
con IncorrectDataList4: element(X, 1 2 3 4 Z.sol, Y);
```

*scalar-variable*

is an *identifier-expression* that refers to a single variable (that is, not to an array of variables).

*variable-list*

is a space-separated list of items, like a *data-list*, except that the items can resolve to variables, variable arrays, linear expressions, and constant numeric values. The items, each of which can be prefixed by an indexing set, conform to one of the following forms of syntax:

- a number
- an *identifier-expression* (but excluding suffixes) that specifies a variable, implicit variable, objective, or numeric parameter
- the name of a variable, implicit variable, objective, or numeric array
- ( *expression* ), which must evaluate to a linear equation or numeric constant

When the value of an element of a *variable-list* refers to a linear equation or numeric constant, it is handled like a solver variable equal to the value.

As an example, the first four constraints in the following statements refer to valid *variable-lists*, whereas the last two do not. Each incorrect constraint is preceded by a comment that explains why the constraint is incorrect.

```
var X{1..3} integer, A integer, B integer;
impvar IX{i in 1..3} = X[i] + i;
num n init 2;
con CorrectVariableList1: alldiff({j in 1..3} X[j]);
con CorrectVariableList2: alldiff(A B);
con CorrectVariableList3: alldiff({j in 1..3} X[j] A B);
con CorrectVariableList4: alldiff(1 n (n+1) IX);
/* Indexing is not distributive in variable lists */
var Y{1..3} integer;
con IncorrectVariableList1: alldiff({j in 1..3} (X[j] Y[j]));
/* Expressions and suffixed ids must appear in parentheses */
con IncorrectVariableList2: alldiff(A.dual);
```

## ALLDIFF Predicate

**ALLDIFF(***variable-list***)**
**ALLDIFFERENT(***variable-list***)**

The ALLDIFF predicate defines an all-different constraint, which defines a unique global constraint on a set of variables that requires all of them to be different from each other. A global constraint is equivalent to a conjunction of elementary constraints.

The syntax of the all-different constraint consists of one part, a *variable-list*, which is defined in the section "Common Syntax Components" on page 200. For example, the statements

```
var X{1..3} integer, A integer, B integer;
con AD1: alldiff({j in 1..3} X[j]);
con AD2: alldiff(A B);
```

are equivalent to

$$\begin{aligned} X[1] &\neq X[2] \text{ AND} \\ X[2] &\neq X[3] \text{ AND} \\ X[1] &\neq X[3] \text{ AND} \\ A &\neq B \end{aligned}$$

To apply the all-different constraint to all the variables, use the problem symbol _VAR_ as follows:

```
con alldiff(_VAR_);
```

For a description of problem symbols, see the section "Problem Symbols" on page 150 in Chapter 5, "The OPTMODEL Procedure."

## ELEMENT Predicate

**ELEMENT(***scalar-variable***,***data-list***,***scalar-variable***)**

The ELEMENT predicate specifies an array element lookup constraint, which enables you to define dependencies (which are not necessarily functional) between variables.

The predicate ELEMENT$(I, L, V)$ sets the variable $V$ to be equal to the $I$th element in the list $L$, where $L = (v_1, \ldots, v_n)$ is a list of values (not necessarily distinct) that the variable $V$ can take. The variable $I$ is the index variable, and its domain is considered to be $[1, n]$. Each time the domain of $I$ is modified, the domain of $V$ is updated, and vice versa.

For example, the following statements use the ELEMENT predicate to determine whether there are squares greater than 1 that are also elements of the Fibonacci sequence:

```
/* Are there any squares > 1 in the Fibonacci sequence? */
proc optmodel;
   num n = 20;
   /* 1 appears twice in the Fibonacci sequence */
   num fib{i in 1..n} = if i < 3 then 1 else fib[i-1] + fib[i-2];

   var IFib integer, ISq integer,
       XFib integer, XSq integer;

   con IsFibAndIsSquare: XFib = XSq;
   /* You can use a numeric array to refer to a list */
   con IdxOfFib: element( IFib, fib, XFib );
   /* You can also build a list from a set iterator */
   con IdxOfSq:  element( ISq,  {i in 2..n} (i * i),  XSq );
   solve;
   print XFib XSq;
quit;
```

An element constraint enforces the propagation rules

$$V = v \Leftrightarrow I \in \{i_1, \ldots, i_m\}$$

where $v$ is a value in the list $L$ and $i_1, \ldots, i_m$ are all the indices in $L$ whose value is $v$.

An element constraint is equivalent to a conjunction of reify and linear constraints. For example, both of the following examples implement the quadratic function, $Y = X^2$:

- Using the ELEMENT predicate:

```
proc optmodel;
   var X >= 1 <= 5 integer, Y >= 1 <= 25 integer;
   num a {i in 1..5} = i^2;
   con Mycon: element(X, a, Y);
   solve;
quit;
```

- Using linear constraints and the REIFY predicate:

```
proc optmodel;
   var X >= 1 <= 5 integer, Y >= 1 <= 25 integer, R {1..5} binary;
   con MyconX {i in 1..5}:
      reify(R[i], X = i);
   con MyconY {i in 1..5}:
      reify(R[i], Y = i^2);
   con SumToOne:
      sum {i in 1..5} R[i] = 1;
   solve;
quit;
```

You can also use element constraints to define positional mappings between two variables. For example, suppose the function $Y = X^2$ is defined on only odd numbers in the interval $[-5, 5]$. You can relate $X$ and $Y$ by using two element constraints and an artificial index variable:

```
var I integer, X integer, Y integer;
/* You can also build a list by providing explicit literals. */
con XsubI: element (I, -5 -3 -1 1 3  5, X);
con YsubI: element (I, 25  9  1 1 9 25, Y);
```

## GCC Predicate

**GCC(***variable-list***,***set-of-numeric-triplets***)**

The GCC predicate specifies a global cardinality constraint (GCC), which sets the minimum and maximum number of times each value can be assigned to a group of variables.

The syntax of the GCC constraint consists of two parts:

*variable-list*
See the section "Common Syntax Components" on page 200.

*set-of-numeric-triplets*

The triplets $< v, l_v, u_v >$ provide, for each value $v$, the minimum $l_v$ and maximum $u_v$ number of times that $v$ can be assigned to the variables in the *variable-list*. The PROC OPTMODEL option INTFUZZ= determines which values are rounded to integers.

Consider the following statements:

```
var X {1..6} >= 1 <= 4 integer;
con Mycon: gcc(X, /<1,1,2>, <2,1,3>, <3,1,3>, <4,2,3>/);
```

These statements specify a constraint that expresses the following requirements about the values of variables $\{X[1], \ldots, X[6]\}$:

- The value 1 must appear at least once but no more than twice.
- The value 2 must appear at least once but no more than three times.
- The value 3 must appear at least once but no more than three times.
- The value 4 must appear at least twice but no more than three times.

The assignment $X[1] = 1, X[2] = 1, X[3] = 2, X[4] = 3, X[5] = 4$, and $X[6] = 4$ satisfies the constraint.

In general, a GCC constraint consists of a set of variables $\{x_1, \ldots, x_n\}$ and, for each value $v$ in $D = \bigcup_{i=1}^{n} \mathrm{Dom}(x_i)$, a pair of numbers $l_v$ and $u_v$. A GCC is satisfied if and only if the number of times that a value $v$ in $D$ is assigned to the variables $x_1, \ldots, x_n$ is at least $l_v$ and at most $u_v$.

Values in the domain of *variable-list* that do not appear in any triplet are unconstrained. They can be assigned to as many of the variables in *variable-list* as needed to produce a feasible solution.

The following statements specify that each of the values in the set $\{1, \ldots, 9\}$ can be assigned to at most one of the variables $X[1], \ldots, X[9]$:

```
var X {1..9} >= 1 <= 9 integer;
con Mycon: gcc(X, setof{i in 1..9} <i,0,1>);
```

Note that the preceding global cardinality constraint is equivalent to the all-different constraint that is expressed as follows:

```
var X {1..9} >= 1 <= 9 integer;
con Mycon: alldiff(X);
```

The global cardinality constraint also provides a convenient way to define disjoint domains for a set of variables. For example, the following syntax limits assignment of the variables $X[1], \ldots, X[9]$ to even numbers between 0 and 10:

```
var X {1..9} >= 0 <= 10 integer;
con Mycon: gcc(X, setof{i in 1..9 by 2} <i,0,0>);
```

## LEXICO Predicate

**LEXICO(***variable-list order-type variable-list***)**

The LEXICO predicate defines a lexicographic ordering constraint, which compares two arrays of the same size from left to right. For example, a standings table in a sports competition is usually ordered lexicographically, with certain attributes (such as wins or points) to the left of others (such as goal difference).

The *order-type* can be either $\leq$, to indicate lexicographically less than or equal to ($\leq_{\text{lex}}$), or $<$, to indicate lexicographically less than ($<_{\text{lex}}$). Given two $n$-tuples $x = (x_1, \ldots, x_n)$ and $y = (y_1, \ldots, y_n)$, the $n$-tuple $x$ is lexicographically less than or equal to $y$ ($x \leq_{\text{lex}} y$) if and only if

$$\left(x_i = y_i \ \forall i = 1, \ldots, n\right) \vee \left(\exists j \text{ with } 1 \leq j \leq n \text{ s.t. } x_i = y_i \ \forall i = 1, \ldots, j-1 \text{ and } x_j < y_j\right)$$

The $n$-tuple $x$ is lexicographically less than $y$ ($x <_{\text{lex}} y$) if and only if $x \leq_{\text{lex}} y$ and $x \neq y$. Equivalently, $x <_{\text{lex}} y$ if and only if

$$\exists j \text{ with } 1 \leq j \leq n \text{ s.t. } x_i = y_i \ \forall i = 1, \ldots, j-1 \text{ and } x_j < y_j$$

Informally, you can think of the lexicographic constraint $\leq_{\text{lex}}$ as sorting the $n$-tuples in alphabetical order. Mathematically, $\leq_{\text{lex}}$ is a partial order on a subset of $n$-tuples, and $<_{\text{lex}}$ is a strict partial order on a subset of $n$-tuples (Brualdi 2010).

For example, you can express the lexicographic constraint $(X[1], \ldots, X[6]) \leq_{\text{lex}} (Y[1], \ldots, Y[6])$ by using a LEXICO predicate as follows:

```
con Mycon: lexico({j in 1..6} X[j] <= {j in 1..6} Y[j]);
```

The assignment $X[1]=1$, $X[2]=2$, $X[3]=2$, $X[4]=1$, $X[5]=2$, $X[6]=5$, $Y[1]=1$, $Y[2]=2$, $Y[3]=2$, $Y[4]=1$, $Y[5]=4$, and $Y[6]=3$ satisfies this constraint because $X[i] = Y[i]$ for $i = 1, \ldots, 4$ and $X[5] < Y[5]$. The fact that $X[6] > Y[6]$ is irrelevant in this ordering.

Lexicographic ordering constraints can be useful for breaking a certain type of symmetry that arises in CSPs and involves matrices of decision variables. Frisch et al. (2002) introduce an optimal algorithm to establish generalized arc consistency (GAC) for the $\leq_{\text{lex}}$ constraint between two vectors of variables.

## PACK Predicate

**PACK(***variable-list*,*data-list*,*variable-list***)**

The PACK predicate specifies a pack constraint, which is used to assign items to bins, subject to the sizes of the items and the capacities of the bins.

For example, suppose you have three bins, whose capacities are 3, 4, and 5, and you have five items of sizes 4, 3, 2, 2, and 1, to be assigned to these three bins. The following statements formulate the problem and find a solution:

```
proc optmodel;
   var SpaceUsed {bin in 1..3} integer >= 0 <= bin + 2;
   var WhichBin {1..5} >= 1 <= 3 integer;
   num itemSize {1..5} = [4 3 2 2 1];
   con pack( WhichBin, itemSize, SpaceUsed );
   solve with clp / findall;
quit;
```

Each row of Table 6.4 represents a solution to the problem. The number in each item column is the number of the bin to which the corresponding item is assigned.

**Table 6.4** Bin Packing Solutions

| WhichBin Variable | | | | |
|---|---|---|---|---|
| **WhichBin[1]** | **WhichBin[2]** | **WhichBin[3]** | **WhichBin[4]** | **WhichBin[5]** |
| 2 | 3 | 3 | 1 | 1 |
| 2 | 3 | 1 | 3 | 1 |
| 2 | 1 | 3 | 3 | 3 |
| 3 | 1 | 2 | 2 | 3 |

When you assign a set of $k$ items to $m$ bins, the item variable $b_i, i \in \{1, \ldots, k\}$ (WhichBin in the preceding example), contains the bin number for the $i$th item. The constant $s_i$ (itemSize in the preceding example) holds the size or weight of the $i$th item. The domain of the load variable $l_j$ (SpaceUsed in the preceding example) constrains the capacity of bin $j$. The value of $l_j$ in the solution is the amount of space used in bin $j$.

**NOTE:** It can be more efficient to assign higher priority to item variables than to load variables, and within the item variables, to assign higher priority to larger items.

## REIFY Predicate

**REIFY(***scalar-variable***,** *linear-constraint***)**

The REIFY predicate specifies a reify constraint, which associates a binary variable with a constraint. The value of the binary variable is 1 or 0, depending on whether the constraint is satisfied or not, respectively. The constraint is said to be reified, and the binary variable is referred to as the *control variable*. Currently, only linear constraints can be reified.

The REIFY predicate provides a convenient mechanism for expressing logical constraints, such as disjunctive and implicative constraints. For example, the disjunctive constraint

$$(3X + 4Y < 20) \vee (5X - 2Y > 50)$$

can be expressed by the following statements:

```
var X integer, Y integer,
    P binary,  Q binary;
con reify(P, 3 * X + 4 * Y < 20);
con reify(Q, 5 * X - 2 * Y > 50);
con AtLeastOneHolds: P + Q >= 1;
```

The binary variable $P$ reifies the linear constraint

$$3X + 4Y < 20$$

The binary variable $Q$ reifies the linear constraint

$$5X - 2Y > 50$$

The following linear constraint enforces the desired disjunction:

$$P + Q \geq 1$$

The following implicative constraint

$$(3X + 4Y < 20) \Rightarrow (5X - 2Y > 50)$$

can be enforced by the linear constraint

$$P \leq Q$$

You can also use the REIFY constraint to express a constraint that involves the absolute value of a variable. For example, the constraint

$$|X| = 5$$

can be expressed by the following statements:

```
var X integer, P binary, Q binary;
con Xis5:      reify(P, X = 5);
con XisMinus5: reify(Q, X = -5);
con OneMustHold: P + Q = 1;
```

# Details: CLP Solver

## Types of CSPs

The CLP solver is a finite-domain constraint programming solver for CSPs. A *standard CSP* is characterized by integer variables, linear constraints, global constraints, and reify constraints. The solver expects only linear, ALLDIFF, ELEMENT, GCC, LEXICO, PACK, and REIFY predicates.

Both PROC OPTMODEL and PROC CLP support standard CSPs. The CLP procedure also supports *scheduling CSP*s, which are characterized by activities, temporal constraints, and resource requirement constraints. For more information about the CLP procedure see *SAS/OR User's Guide: Constraint Programming*.

## Techniques for Solving CSPs

Several techniques for solving CSPs are available. Kumar (1992) and Tsang (1993) present a good overview of these techniques. It should be noted that the satisfiability problem (SAT) (Garey and Johnson 1979) can be regarded as a CSP. Consequently, most problems in this class are nondeterministic polynomial-time complete (NP-complete) problems, and a backtracking search mechanism is an important technique for solving them (Floyd 1967).

One of the most popular tree search mechanisms is chronological backtracking. However, a chronological backtracking approach is not very efficient because conflicts are detected late; that is, the approach is oriented toward *recovering* from failures rather than *avoiding* them to begin with. The search space is reduced only after a failure is detected, and the performance of this technique is drastically reduced as the problem size increases. Another drawback of using chronological backtracking, for the same reason, is encountering repeated failures, sometimes called "thrashing." The presence of late detection and "thrashing" has led researchers to develop consistency techniques that can achieve superior pruning of the search tree. This strategy uses constraints actively, rather than passively.

## Constraint Propagation

A more efficient technique than backtracking is constraint propagation, which uses consistency techniques to effectively prune the domains of variables. Consistency techniques are based on the idea of a priori pruning, which uses the constraint to reduce the domains of the variables. Consistency techniques are also known as relaxation algorithms (Tsang 1993), and the process is also called problem reduction, domain filtering, or pruning.

One of the earliest applications of consistency techniques was in the AI field to solve the scene labeling problem, which required recognizing objects in three-dimensional space by interpreting two-dimensional line drawings of the object. The Waltz filtering algorithm (Waltz 1975) analyzes line drawings by systematically labeling the edges and junctions while maintaining consistency between the labels.

Constraint propagation is characterized by the extent of propagation (also called the level of consistency) and whether domain propagation or interval propagation is the domain pruning scheme that is followed. In practice, interval propagation is preferred because of its lower computational costs. This mechanism is discussed in detail in Van Hentenryck (1989). However, constraint propagation is not a complete solution technique and needs to be complemented by a search technique in order to ensure success (Kumar 1992).

## Finite-Domain Constraint Programming

Finite-domain constraint programming is an effective and complete solution technique that embeds incomplete constraint propagation techniques into a nondeterministic backtracking search mechanism that is implemented as follows: Whenever a node is visited, constraints are propagated to attain a desired level of consistency. If the domain of each variable reduces to a singleton set, the node represents a solution to the CSP. If the domain of a variable becomes empty, the node is pruned. Otherwise a variable is selected, its domain is distributed, and a new set of CSPs is generated, each of which is a child node of the current node. Several factors play a role in determining the outcome of this mechanism, such as the extent of propagation (or level of consistency enforced), the variable selection strategy, and the variable assignment or domain distribution strategy.

For example, the lack of any propagation reduces this technique to a simple generate-and-test approach, whereas performing consistency checking using variables that are already selected reduces this approach to chronological backtracking, one of the systematic search techniques. These are also known as look-back schemas, because they share the disadvantage of late conflict detection. Look-ahead schemas, on the other hand, work to prevent future conflicts. Some popular examples of look-ahead schemas, in increasing degree of consistency level, are forward checking (FC), partial look ahead (PLA), and full look ahead (LA) (Kumar 1992). Forward checking enforces consistency between the current variable and future variables; PLA and LA extend this even further to pairs of not yet instantiated variables.

Two important consequences of this technique are that inconsistencies are discovered early and that the current set of alternatives that are coherent with the existing partial solution is dynamically maintained. These consequences are powerful enough to prune large parts of the search tree, thereby reducing the "combinatorial explosion" of the search process. However, although constraint propagation at each node results in fewer nodes in the search tree, the processing at each node is more expensive. The ideal scenario is to strike a balance between the extent of propagation and the subsequent computation cost.

Variable selection is another strategy that can affect the solution process. The order in which variables are chosen for instantiation can have a substantial impact on the complexity of the backtrack search. Several heuristics have been developed and analyzed for selecting variable ordering. One of the most common ones is a dynamic heuristic based on the *fail-first principle* (Haralick and Elliott 1980), which selects the variable

whose domain has minimal size. Subsequent analysis of this heuristic by several researchers has validated this strategy as providing substantial improvement for a significant class of problems. Another popular strategy is to instantiate the most constrained variable first. Both these strategies are based on the principle of selecting the variable most likely to fail and detecting such failures as early as possible.

The domain distribution strategy for a selected variable is yet another area that can influence the performance of a backtracking search. However, good value-ordering heuristics are expected to be very problem-specific (Kumar 1992).

### Consistency Techniques

The CLP solver features a full look-ahead algorithm for standard CSPs that follows a strategy of maintaining a version of generalized arc consistency that is based on the AC-3 consistency routine (Mackworth 1977). This strategy maintains consistency between the selected variables and the unassigned variables and also maintains consistency between unassigned variables.

### Selection Strategy

A search algorithm for CSPs searches systematically through the possible assignments of values to variables. The order in which a variable is selected can be based on a *static* ordering, which is determined before the search begins, or on a *dynamic* ordering, in which the choice of the next variable depends on the current state of the search. The VARSELECT= option in the SOLVE statement defines the variable selection strategy for a standard CSP. The default strategy is the dynamic MINR strategy, which selects the variable that has the smallest range.

### Assignment Strategy

After a variable is selected, the assignment strategy dictates the value that is assigned to it. For variables, the assignment strategy is specified in the VARASSIGN= option in the SOLVE statement. The default assignment strategy selects the minimum value from the domain of the selected variable.

## Differences between PROC OPTMODEL and PROC CLP

You can invoke the CLP solver from PROC OPTMODEL by using any of the predicates that are defined in the standard mode of PROC CLP. The *standard mode* gives you access to all-different, element, GCC, linear, pack, and reify constraints.

To replicate the FOREACH statement that PROC CLP supports, you can use PROC OPTMODEL's expressions and iteration machinery. For an example, see the Eight Queens example in the "Getting Started: CLP Solver" on page 192. For more information about the FOREACH predicate, see *SAS/OR User's Guide: Constraint Programming*.

In addition to the predicates that are defined in this chapter, PROC CLP provides several constraints and capabilities that simplify the modeling of scheduling-oriented CSPs. For more information about those statements, see the section "Details: CLP Procedure" (Chapter 3, *SAS/OR User's Guide: Constraint Programming*).

PROC OPTMODEL has different syntax and semantics for variable declarations:

- Because all CLP variables are discrete, you must declare every variable that a CLP model uses as integer or binary.
- The default variable bounds in PROC OPTMODEL are $-\infty$ to $\infty$. The default lower bound in PROC CLP is 0. Thus, to replicate the behavior of PROC CLP, you must explicitly add a lower bound of 0 to the variable declaration.

## Macro Variable _OROPTMODEL_

The OPTMODEL procedure always creates and initializes a SAS macro variable called _OROPTMODEL_, which contains a character string. The variable contains information about the execution of the most recently invoked solver.

Each keyword and value pair in _OROPTMODEL_ also appears in two other places: the PROC OPTMODEL automatic arrays _OROPTMODEL_NUM_ and _OROPTMODEL_STR_; and the ODS tables ProblemSummary and SolutionSummary, which appear after a SOLVE statement, unless you set the PRINTLEVEL= option to NONE. You can use these variables to obtain details about the solution.

After the solver is called, the various keywords in the variable are interpreted as follows.

**STATUS**
indicates the solver status at termination. It can take one of the following values:

| | |
|---|---|
| OK | The solver terminated normally. |
| SYNTAX_ERROR | The use of syntax is incorrect. |
| DATA_ERROR | The input data are inconsistent. |
| OUT_OF_MEMORY | Insufficient memory was allocated to the procedure. |
| IO_ERROR | A problem in reading or writing of data has occurred. |
| SEMANTIC_ERROR | An evaluation error, such as an invalid operand type, has occurred. |
| ERROR | The status cannot be classified into any of the preceding categories. |

**SOLUTION_STATUS**
indicates the solution status at termination. It can take one of the following values:

| | |
|---|---|
| ABORT_NOSOL | The solver was stopped by the user and did not find a solution. |
| ABORT_SOL | The solver was stopped by the user but still found a solution. |
| ALL_SOLUTIONS | All solutions were found. |
| BAD_PROBLEM_TYPE | The problem type is not supported by the solver. |
| CONDITIONAL_OPTIMAL | The optimality of the solution cannot be proven. |
| ERROR | The algorithm encountered an error. |
| FAIL_NOSOL | The solver stopped because of errors and did not find a solution. |

| | |
|---|---|
| FAIL_SOL | The solver stopped because of errors but still found a solution. |
| INFEASIBLE | The problem is infeasible. |
| INTERRUPTED | The solver was interrupted by the system or the user before completing its work. |
| OK | The algorithm terminated normally. |
| OPTIMAL | The solution is optimal. |
| OUTMEM_NOSOL | The solver ran out of memory and either did not find a solution or failed to output the solution because of insufficient memory. |
| OUTMEM_SOL | The solver ran out of memory but still found a solution. |
| SOLUTION_LIM | The solver reached the maximum number of solutions specified in the MAXSOLNS= option. |
| TIME_LIM_NOSOL | The solver reached the execution time limit specified in the MAXTIME= option and did not find a solution. |
| TIME_LIM_SOL | The solver reached the execution time limit specified in the MAXTIME= option and found a solution. |
| UNBOUNDED | The problem is unbounded. |

**ALGORITHM**
indicates the algorithm that produces the solution data in the macro variable. This term appears only when STATUS=OK. It can take only one value in the CLP solver: CLP, which indicates that the constraint satisfaction algorithm produced the solution data.

**OBJECTIVE**
indicates the objective value that the solver obtained at termination. If a problem does not have an explicit objective, the value of this keyword in the _OROPTMODEL_ macro variable is missing (.).

**PRESOLVE_TIME**
indicates the time (in seconds) that the algorithm used for preprocessing. You can use the TIMETYPE= option to select real time or CPU time.

**SOLUTION_TIME**
indicates the time (in seconds) that the algorithm used to perform iterations to solve the problem. You can use the TIMETYPE= option to select real time or CPU time.

**SOLUTIONS_FOUND**
indicates the number of solutions found, which might be 0 if the problem is infeasible. This keyword is always present in the solution status when you call the CLP solver. The value might not be the total number of solutions possible (for example, if the solver reached its time limit).

# Examples: CLP Solver

## Example 6.1: Logic-Based Puzzles

Many logic-based puzzles can be formulated as CSPs. Several such puzzles are shown in this example.

### Sudoku

Sudoku is a logic-based, combinatorial number-placement puzzle that uses a partially filled 9 × 9 grid. The objective is to fill the grid with the digits 1 to 9 so that each column, each row, and each of the nine 3 × 3 blocks contains only one of each digit. Figure 6.1.1 shows an example of a sudoku grid.

**Output 6.1.1** Example of an Unsolved Sudoku Grid

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | 5 | | | 7 | | | 1 |
| | 7 | | | 9 | | | 3 | |
| | | | 6 | | | | | |
| | | 3 | | | 1 | | | 5 |
| | 9 | | | 8 | | | 2 | |
| 1 | | | 2 | | | 4 | | |
| | | 2 | | | 6 | | | 9 |
| | | | | 4 | | | 8 | |
| 8 | | | 1 | | | 5 | | |

This example illustrates the use of the all-different constraint to solve the preceding sudoku problem. The data set Indata contains the partially filled values for the grid.

```
data Indata;
   input C1-C9;
   datalines;
. . 5 . . 7 . . 1
. 7 . . 9 . . 3 .
. . . 6 . . . . .
. . 3 . . 1 . . 5
. 9 . . 8 . . 2 .
1 . . 2 . . 4 . .
. . 2 . . 6 . . 9
. . . . 4 . . 8 .
8 . . 1 . . 5 . .
;
```

Let the variable $X_{ij}(i = 1, \ldots, 9, j = 1, \ldots, 9)$ represent the value of cell $(i, j)$ in the grid. The domain of each of these variables is $[1, 9]$. When cell $(i, j)$ is not missing in the data set, $X_{ij}$ is fixed to that value.

Three sets of all-different constraints specify the required rules for each row, each column, and each of the $3 \times 3$ blocks. The RowCon constraint forces all values in row $i$ to be different, the ColumnCon constraint forces all values in column $j$ to be different, and the BlockCon constraint forces all values in each block to be different.

The following statements express the preceding constraints in PROC OPTMODEL and solve the sudoku puzzle:

```
proc optmodel;
   /* Declare variables */
   set ROWS = 1..9;
   set COLS = ROWS; /* Use an alias for convenience and clarity */
   var X {ROWS, COLS} >= 1 <= 9 integer;

   /* Nine row constraints */
   con RowCon {i in ROWS}:
      alldiff({j in COLS} X[i,j]);

   /* Nine column constraints */
   con ColCon {j in COLS}:
      alldiff({i in ROWS} X[i,j]);

   /* Nine 3x3 block constraints */
   con BlockCon {s in 0..2, t in 0..2}:
      alldiff({i in 3*s+1..3*s+3, j in 3*t+1..3*t+3} X[i,j]);

   /* Fix variables to cell values */
   /* X[i,j] = c[i,j] if c[i,j] is not missing */
   num c {ROWS, COLS};
   read data indata into [_N_] {j in COLS} <c[_N_,j]=col('C'||j)>;
   for {i in ROWS, j in COLS: c[i,j] ne .}
      fix X[i,j] = c[i,j];

   solve;
quit;
```

Output 6.1.2 shows the solution.

**Output 6.1.2** Solution of the Sudoku Grid

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 9 | 8 | 5 | 3 | 2 | 7 | 6 | 4 | 1 |
| 6 | 7 | 1 | 5 | 9 | 4 | 2 | 3 | 8 |
| 3 | 2 | 4 | 6 | 1 | 8 | 9 | 5 | 7 |
| 2 | 4 | 3 | 7 | 6 | 1 | 8 | 9 | 5 |
| 5 | 9 | 7 | 4 | 8 | 3 | 1 | 2 | 6 |
| 1 | 6 | 8 | 2 | 5 | 9 | 4 | 7 | 3 |
| 4 | 5 | 2 | 8 | 3 | 6 | 7 | 1 | 9 |
| 7 | 1 | 6 | 9 | 4 | 5 | 3 | 8 | 2 |
| 8 | 3 | 9 | 1 | 7 | 2 | 5 | 6 | 4 |

## Pi Day Sudoku

The basic structure of the classical sudoku problem can easily be extended to formulate more complex puzzles. One such example is the Pi Day sudoku puzzle.

Pi Day is a celebration of the number $\pi$ that occurs every March 14. In honor of Pi Day, Brainfreeze Puzzles (Riley and Taalman 2008) celebrates this day with a special $12 \times 12$ grid sudoku puzzle. The 2008 Pi Day sudoku puzzle is shown in Figure 6.1.3.

**Output 6.1.3** Pi Day Sudoku 2008

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 3 | | | 1 | 5 | 4 | | | 1 | | 9 | 5 |
| | 1 | | | 3 | | | | | 1 | 3 | 6 |
| | | 4 | | | 3 | | 8 | | | 2 | |
| 5 | | | 1 | | | 9 | 2 | 5 | | | 1 |
| | 9 | | | 5 | | | 5 | | | | |
| 5 | 8 | 1 | | | 9 | | | 3 | | 6 | |
| | 5 | | 8 | | | 2 | | | 5 | 5 | 3 |
| | | | | 5 | | | 6 | | | 1 | |
| 2 | | | 5 | 1 | 5 | | | 5 | | | 9 |
| | 6 | | | 4 | | 1 | | | 3 | | |
| 1 | 5 | 1 | | | | | 5 | | | 5 | |
| 5 | 5 | | 4 | | | 3 | 1 | 6 | | | 8 |

The rules for this puzzle are a little different from the rules for standard sudoku:

1. Rather than using regular $3 \times 3$ blocks, this puzzle uses jigsaw regions such that highlighted regions in the middle resemble the Greek letter $\pi$. Each jigsaw region consists of 12 contiguous cells.
2. The first 12 digits of $\pi$ are used instead of the digits 1–9. Each row, column, and jigsaw region contains the first 12 digits of $\pi$ (314159265358) in some order. In other words, there are no 7s; one each of 2, 4, 6, 8, and 9; two each of 1 and 3; and three 5s.

To generalize the original sudoku model:

1. Replace the expression that calculates the starting and ending cells of a region by an array that maps each cell to one region.
2. Replace the all-different constraints with GCC constraints. GCC constraints describe how often each value can be assigned to a set of variables. Conceptually, an all-different constraint is a specialized GCC constraint in which both the lower bound and the upper bound of every value is 1.

The data set Raw contains the partially filled values for the grid. It contains missing values where the cell does not yet contain a number.

```
data Raw;
   input C1-C12;
   datalines;
3  .  .  1  5  4  .  .  1  .  9  5
.  1  .  .  3  .  .  .  .  1  3  6
.  .  4  .  .  3  .  8  .  .  2  .
5  .  .  1  .  .  9  2  5  .  .  1
.  9  .  .  5  .  .  5  .  .  .  .
5  8  1  .  .  9  .  .  3  .  6  .
.  5  .  8  .  .  2  .  .  5  5  3
.  .  .  .  5  .  .  6  .  .  1  .
2  .  .  5  1  5  .  .  5  .  .  9
.  6  .  .  4  .  1  .  .  3  .  .
1  5  1  .  .  .  .  5  .  .  5  .
5  5  .  4  .  .  3  1  6  .  .  8
;
```

The following statements define the GCC constraints in order to find all solutions of the Pi Day sudoku 2008 puzzle:

```
proc optmodel;
   set ROWS = 1..12;
   /* These declarations are inexpensive and improve clarity: */
   set COLS = ROWS, REGIONS = ROWS, CELLS = ROWS cross COLS;

   /* specify a 12x12 array of region identifiers.
      The spacing is just to make the regions easier to visualize. */
   num region{CELLS} = [
       1  1  1    2  2  2  2  2  2    3  3  3
       1  1  1    2  2  2  2  2  2    3  3  3
       1  1   4  4  4  4    5  5  5  5   3  3
       1  1   4  4  4  4    5  5  5  5   3  3
       1  1   4  4  4  4    5  5  5  5   3  3
       6  6  6   7  7   8  8   9  9  10 10 10
       6  6  6   7  7   8  8   9  9  10 10 10
       6  6  6   7  7   8  8   9  9  10 10 10
       6  6  6   7  7   8  8   9  9  10 10 10
      11 11 11   7  7   8  8   9  9  12 12 12
      11 11 11   7  7   8  8   9  9  12 12 12
      11 11 11 11 11 11     12 12 12 12 12 12 ];

   /* Each area must contain two 1's, two 3's, three 5's, no 7's,
      and one for each of other values from 1 to 9. */
                  /* 1 2 3 4 5 6 7 8 9 */
   num nTimes{1..9} = [2 1 2 1 3 1 0 1 1];
   /* For convenience, create a triplet set version of nTimes.
      In this model, GCC's lower and upper bounds are the same. */
   set N_TIMES = setof{ni in 1..9} <ni,nTimes[ni],nTimes[ni]>;

   /* The number assigned to the ith row and jth column. */
   var X {CELLS} >= 1 <= 9 integer;

   /* X[i,j] = c[i,j] if c[i,j] is not missing */
   num c {CELLS};
   read data raw into [_N_] {j in COLS} <c[_N_,j]=col('C'||j)>;
   for {<i,j> in CELLS: c[i,j] ne .}
      fix X[i,j] = c[i,j];

   con RowCon {i in ROWS}:
      gcc({j in COLS} X[i,j], N_TIMES);

   con ColCon {j in COLS}:
      gcc({i in ROWS} X[i,j], N_TIMES);

   con RegionCon {ri in REGIONS}:
      gcc({<i,j> in CELLS: region[i,j] = ri} X[i,j], N_TIMES);

   solve;
   /* Replicate typical PROC CLP output from PROC OPTMODEL arrays */
   create data pdsout from
      {<i,j> in ROWS cross COLS}<col('X_'||i||'_'||j)=X[i,j]>;
quit;
```

The only solution of the 2008 Pi Day sudoku puzzle is shown in Output 6.1.4.

**Output 6.1.4** Solution to Pi Day Sudoku 2008

**Pi Day Sudoku 2008**

| Obs | C1 | C2 | C3 | C4 | C5 | C6 | C7 | C8 | C9 | C10 | C11 | C12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **1** | 3 | 2 | 5 | 1 | 5 | 4 | 6 | 3 | 1 | 8 | 9 | 5 |
| **2** | 4 | 1 | 5 | 2 | 3 | 8 | 5 | 9 | 5 | 1 | 3 | 6 |
| **3** | 6 | 1 | 4 | 5 | 9 | 3 | 5 | 8 | 3 | 1 | 2 | 5 |
| **4** | 5 | 3 | 3 | 1 | 8 | 5 | 9 | 2 | 5 | 6 | 4 | 1 |
| **5** | 8 | 9 | 2 | 6 | 5 | 1 | 1 | 5 | 4 | 3 | 3 | 5 |
| **6** | 5 | 8 | 1 | 5 | 2 | 9 | 4 | 3 | 3 | 5 | 6 | 1 |
| **7** | 1 | 5 | 3 | 8 | 1 | 6 | 2 | 4 | 9 | 5 | 5 | 3 |
| **8** | 9 | 4 | 5 | 3 | 5 | 1 | 5 | 6 | 8 | 2 | 1 | 3 |
| **9** | 2 | 3 | 6 | 5 | 1 | 5 | 3 | 1 | 5 | 4 | 8 | 9 |
| **10** | 3 | 6 | 8 | 9 | 4 | 5 | 1 | 5 | 1 | 3 | 5 | 2 |
| **11** | 1 | 5 | 1 | 3 | 6 | 3 | 8 | 5 | 2 | 9 | 5 | 4 |
| **12** | 5 | 5 | 9 | 4 | 3 | 2 | 3 | 1 | 6 | 5 | 1 | 8 |

The corresponding completed grid is shown in Figure 6.1.5.

**Output 6.1.5** Solution to Pi Day Sudoku 2008

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 3 | 2 | 5 | 1 | 5 | 4 | 6 | 3 | 1 | 8 | 9 | 5 |
| 4 | 1 | 5 | 2 | 3 | 8 | 5 | 9 | 5 | 1 | 3 | 6 |
| 6 | 1 | 4 | 5 | 9 | 3 | 5 | 8 | 3 | 1 | 2 | 5 |
| 5 | 3 | 3 | 1 | 8 | 5 | 9 | 2 | 5 | 6 | 4 | 1 |
| 8 | 9 | 2 | 6 | 5 | 1 | 1 | 5 | 4 | 3 | 3 | 5 |
| 5 | 8 | 1 | 5 | 2 | 9 | 4 | 3 | 3 | 5 | 6 | 1 |
| 1 | 5 | 3 | 8 | 1 | 6 | 2 | 4 | 9 | 5 | 5 | 3 |
| 9 | 4 | 5 | 3 | 5 | 1 | 5 | 6 | 8 | 2 | 1 | 3 |
| 2 | 3 | 6 | 5 | 1 | 5 | 3 | 1 | 5 | 4 | 8 | 9 |
| 3 | 6 | 8 | 9 | 4 | 5 | 1 | 5 | 1 | 3 | 5 | 2 |
| 1 | 5 | 1 | 3 | 6 | 3 | 8 | 5 | 2 | 9 | 5 | 4 |
| 5 | 5 | 9 | 4 | 3 | 2 | 3 | 1 | 6 | 5 | 1 | 8 |

## Magic Square

A magic square is an arrangement of the distinct positive integers from 1 to $n^2$ in an $n \times n$ matrix such that the sum of the numbers of any row, any column, or any main diagonal is the same number, known as the magic constant. The magic constant of a normal magic square depends only on $n$ and has the value $n(n^2+1)/2$.

This example illustrates the use of the MINRMAXC selection strategy, which is controlled by the VARSELECT= option.

```
%macro magic(n);
   proc optmodel;
      num n = &n;
      /* magic constant */
      num sum = n*(n^2+1)/2;
      set ROWS = 1..n;
      set COLS = 1..n;

      /* X[i,j] = entry (i,j) */
      var X {ROWS, COLS} >= 1 <= n^2 integer;

      /* row sums */
      con RowCon {i in ROWS}:
         sum {j in COLS} X[i,j] = sum;

      /* column sums */
      con ColCon {j in COLS}:
         sum {i in ROWS} X[i,j] = sum;

      /* diagonal: upper left to lower right */
      con DiagCon:
         sum {i in ROWS} X[i,i] = sum;

      /* diagonal: upper right to lower left */
      con AntidiagCon:
         sum {i in ROWS} X[n+1-i,i] = sum;

      /* symmetry-breaking */
      con BreakRowSymmetry:
         X[1,1] + 1 <= X[n,1];
      con BreakDiagSymmetry:
         X[1,1] + 1 <= X[n,n];
      con BreakAntidiagSymmetry:
         X[1,n] + 1 <= X[n,1];

      con alldiff(X);

      solve with CLP / varselect=minrmaxc maxtime=3;
   quit;
%mend magic;

%magic(7)
```

The solution is displayed in Output 6.1.6.

**Output 6.1.6** Solution of the Magic Square

| | | | | | | |
|---|---|---|---|---|---|---|
| 1 | 39 | 24 | 40 | 31 | 38 | 2 |
| 43 | 4 | 34 | 41 | 42 | 5 | 6 |
| 18 | 20 | 23 | 29 | 30 | 22 | 33 |
| 36 | 37 | 25 | 3 | 7 | 35 | 32 |
| 14 | 19 | 44 | 13 | 47 | 12 | 26 |
| 17 | 45 | 9 | 21 | 8 | 48 | 27 |
| 46 | 11 | 16 | 28 | 10 | 15 | 49 |

## Example 6.2: Alphabet Blocks Problem

This example illustrates the use of the global cardinality constraint (GCC). The alphabet blocks problem consists of finding an arrangement of letters on four alphabet blocks. Each alphabet block has a single letter on each of its six sides. Collectively, the four blocks contain every letter of the alphabet except Q and Z. By arranging the blocks in various ways, the following words should be spelled out: BAKE, ONYX, ECHO, OVAL, GIRD, SMUG, JUMP, TORN, LUCK, VINY, LUSH, and WRAP.

You can formulate this problem as a CSP by representing each of the 24 letters as an integer variable. The domain of each variable is the set $\{1, 2, 3, 4\}$, which represents block1 through block4. The assignment A = 1 indicates that the letter A is on a side of block1. Because each block has six sides, each value $v$ in $\{1, 2, 3, 4\}$ must be assigned to exactly six variables so that each side of a block has a letter on it. This restriction can be formulated as a global cardinality constraint over all 24 variables, with common lower and upper bounds set equal to 6.

Moreover, in order to spell all the words listed previously, the four letters in each of the 12 words must be on different blocks. Another GCC statement that specifies 12 global cardinality constraints enforces these conditions. You can also formulate these restrictions by using 12 all-different constraints. Finally, four FIX statements break the symmetries that blocks are interchangeable. These constraints preset the blocks that contain the letters B, A, K, and E as block1, block2, block3, and block4, respectively.

The complete representation of the problem is as follows:

```
proc optmodel;
   /* Each letter except Q and Z is represented with a variable. */
   /* The domain of each variable is the set of 4 blocks,        */
   /*   or {1, 2, 3, 4} for short.                               */
   set LETTERS = / A B C D E F G H I J K L M N O P R S T U V W X Y /;
   var Block {LETTERS} integer >= 1 <= 4;
   set BLOCKS = 1..4;

   /* There are exactly 6 letters on each alphabet block */
   con SixLettersPerBlock:
      gcc(Block, setof {b in BLOCKS} <b,6,6>);

   /* The letters in each word must be on different blocks. */
   set WORDS = / BAKE ONYX ECHO OVAL GIRD SMUG JUMP TORN LUCK VINY LUSH WRAP /;
   con CanSpell {w in WORDS}:
      gcc({k in 1..length(w)} Block[char(w,k)], setof {b in BLOCKS} <b,0,1>);

   /* Note 2: These restrictions can also be enforced by ALLDIFF constraints:
   con CanSpellv2 {w in WORDS}:
      alldiff({k in 1..length(w)} Block[char(w,k)]);
   */

   /* Breaking the symmetry that blocks can be interchanged by setting
      the block that contains the letter B as block1, the block that
      contains the letter A as block2, etc. */
   for {k in 1..length('BAKE')} fix Block[char('BAKE',k)] = k;

   solve;
   print Block;
quit;
```

The solution to this problem is shown in Output 6.2.1.

**Output 6.2.1** Solution to Alphabet Blocks Problem

**The OPTMODEL Procedure**

| Problem Summary | |
|---|---|
| **Objective Sense** | Minimization |
| **Objective Function** | (0) |
| **Objective Type** | Constant |
| | |
| **Number of Variables** | 24 |
| **Bounded Above** | 0 |
| **Bounded Below** | 0 |
| **Bounded Below and Above** | 20 |
| **Free** | 0 |
| **Fixed** | 4 |
| **Binary** | 1 |
| **Integer** | 23 |
| | |
| **Number of Constraints** | 13 |
| **Linear LE (<=)** | 0 |
| **Linear EQ (=)** | 0 |
| **Linear GE (>=)** | 0 |
| **Linear Range** | 0 |
| **Alldiff** | 0 |
| **Element** | 0 |
| **GCC** | 13 |
| **Lexico (<=)** | 0 |
| **Lexico (<)** | 0 |
| **Pack** | 0 |
| **Reify** | 0 |
| | |
| **Constraint Coefficients** | 0 |

| Solution Summary | |
|---|---|
| **Solver** | CLP |
| **Variable Selection** | MINR |
| **Objective Function** | (0) |
| **Solution Status** | Solution Limit Reached |
| **Objective Value** | 0 |
| | |
| **Solutions Found** | 1 |
| **Presolve Time** | 0.00 |
| **Solution Time** | 0.00 |

**Output 6.2.1** *continued*

| [1] | Block |
|---|---|
| A | 2 |
| B | 1 |
| C | 2 |
| D | 2 |
| E | 4 |
| F | 1 |
| G | 4 |
| H | 3 |
| I | 1 |
| J | 2 |
| K | 3 |
| L | 4 |
| M | 3 |
| N | 2 |
| O | 1 |
| P | 4 |
| R | 3 |
| S | 2 |
| T | 4 |
| U | 1 |
| V | 3 |
| W | 1 |
| X | 3 |
| Y | 4 |

## Example 6.3: Work-Shift Scheduling Problem

This example illustrates the use of the GCC constraint to find a feasible solution to a work-shift scheduling problem and then the use of the element constraint to incorporate cost information in order to find a minimum-cost schedule.

Six workers (Alan, Bob, Juanita, Mike, Ravi, and Aisha) are to be assigned to three working shifts. The first shift needs at least one and at most four people; the second shift needs at least two and at most three people; and the third shift needs exactly two people. Alan cannot work on the first shift; Bob can work only on the third shift. The others can work on any shift. The objective is to find a feasible assignment for this problem.

You can model the minimum and maximum shift requirements by using a GCC constraint and formulate the problem as a standard CSP. The variables W[1],. . . ,W[6] identify the shift to which each of the six workers is assigned: Alan, Bob, Juanita, Mike, Ravi, and Aisha.

```
proc optmodel;
   /*  Six workers (Alan, Bob, Juanita, Mike, Ravi and Aisha)
       are to be assigned to 3 working shifts.            */
   set WORKERS = 1..6;
   var W {WORKERS} integer >= 1 <= 3;

   /* The first shift needs at least 1 and at most 4 people;
```

```
      the second shift needs at least 2 and at most 3 people;
      and the third shift needs exactly 2 people. */
   con ShiftNeeds:
      gcc(W, /<1,1,4>,<2,2,3>,<3,2,2>/);

   /* Alan doesn't work on the first shift. */
   con Alan:
      W[1] ne 1;

   /* Bob works only on the third shift. */
   fix W[2] = 3;

   solve;
   print W;
quit;
```

The resulting assignment is shown in Output 6.3.1.

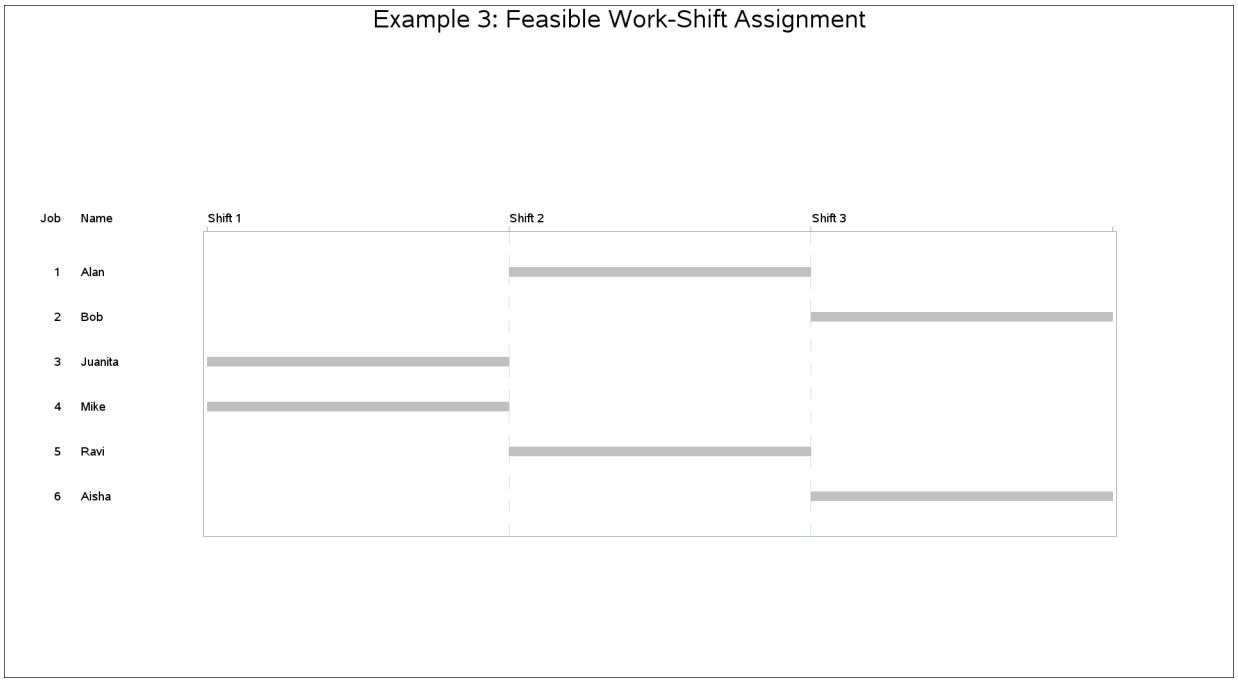
**Output 6.3.1** Solution to Work-Shift Scheduling Problem

**Solution to Work-Shift Scheduling Problem**

| Obs | W1 | W2 | W3 | W4 | W5 | W6 |
|---|---|---|---|---|---|---|
| **1** | 2 | 3 | 1 | 1 | 2 | 3 |

A Gantt chart of the corresponding schedule is displayed in Output 6.3.2.

**Output 6.3.2** Work-Shift Schedule



Now suppose that every work-shift assignment has a cost associated with it and that the objective of interest is to determine the schedule that has the lowest cost.

The costs of assigning the workers to the different shifts are shown in Table 6.5. A dash ("-") in position $(i, j)$ indicates that worker $i$ cannot work on shift $j$.

**Table 6.5** Costs of Assigning Workers to Shifts

| | Shift 1 | Shift 2 | Shift 3 |
|---|---|---|---|
| Alan | - | 12 | 10 |
| Bob | - | - | 6 |
| Juanita | 16 | 8 | 12 |
| Mike | 10 | 6 | 8 |
| Ravi | 6 | 6 | 8 |
| Aisha | 12 | 4 | 4 |

Based on the cost structure in Table 6.5, the previously derived schedule has a cost of \$54. The objective now is to determine the optimal schedule—one that results in the minimum cost.

Let the variable $C_i$ represent the cost of assigning worker $i$ to a shift. This variable is shift-dependent and is given a high value (for example, 100) if the worker cannot be assigned to a shift. The costs can also be interpreted as preferences if desired. You can use an element constraint to associate the cost $C_i$ with the shift assignment for each worker. For example, $C_1$, the cost of assigning Alan to a shift, can be determined by the constraint ELEMENT$(W_1, (100, 12, 10), C_1)$.

By adding a linear constraint, $\sum_{i=1}^{n} C_i \leq \text{obj}$, you can limit the solutions to feasible schedules that cost no more than obj. Although an upper bound of \$100 is used in this example, it would suffice to use an upper bound of \$54, the cost of the feasible schedule that was determined earlier.

```
proc optmodel;
   /* Six workers (Alan, Bob, Juanita, Mike, Ravi and Aisha)
      are to be assigned to 3 working shifts.  */
   set WORKERS = 1..6;
   set SHIFTS  = 1..3;
   var W {WORKERS} integer >= 1 <= 3;
   var C {WORKERS} integer >= 1 <= 100;

   /* The first shift needs at least 1 and at most 4 people;
      the second shift needs at least 2 and at most 3 people;
      and the third shift needs exactly 2 people. */
   con GccCon:
      gcc(W, /<1,1,4>,<2,2,3>,<3,2,2>/);

   /* Alan doesn't work on the first shift. */
   con Alan:
      W[1] ne 1;

   /* Bob works only on the third shift. */
   fix W[2] = 3;

   /* Specify the costs of assigning the workers to the shifts.
      Use 100 (a large number) to indicate an assignment
```

```
      that is not possible.*/
   num a {WORKERS, SHIFTS} = [
      100,  12, 10,
      100, 100,  6,
       16,   8, 12
       10,   6,  8
        6,   6,  8
       12,   4,  4
   ];
   con ElementCon {j in WORKERS}:
      element(W[j], {k in SHIFTS} a[j,k], C[j]);

   /* Minimize total cost. */
   min TotalCost = sum {j in WORKERS} C[j];
   con TotalCost_bounds:
      1 <= TotalCost <= 100;

   solve;
   print W;
   create data clpout from
      {j in WORKERS} <col('W'||j)=W[j]> {j in WORKERS} <col('C'||j)=C[j]>;
quit;
```

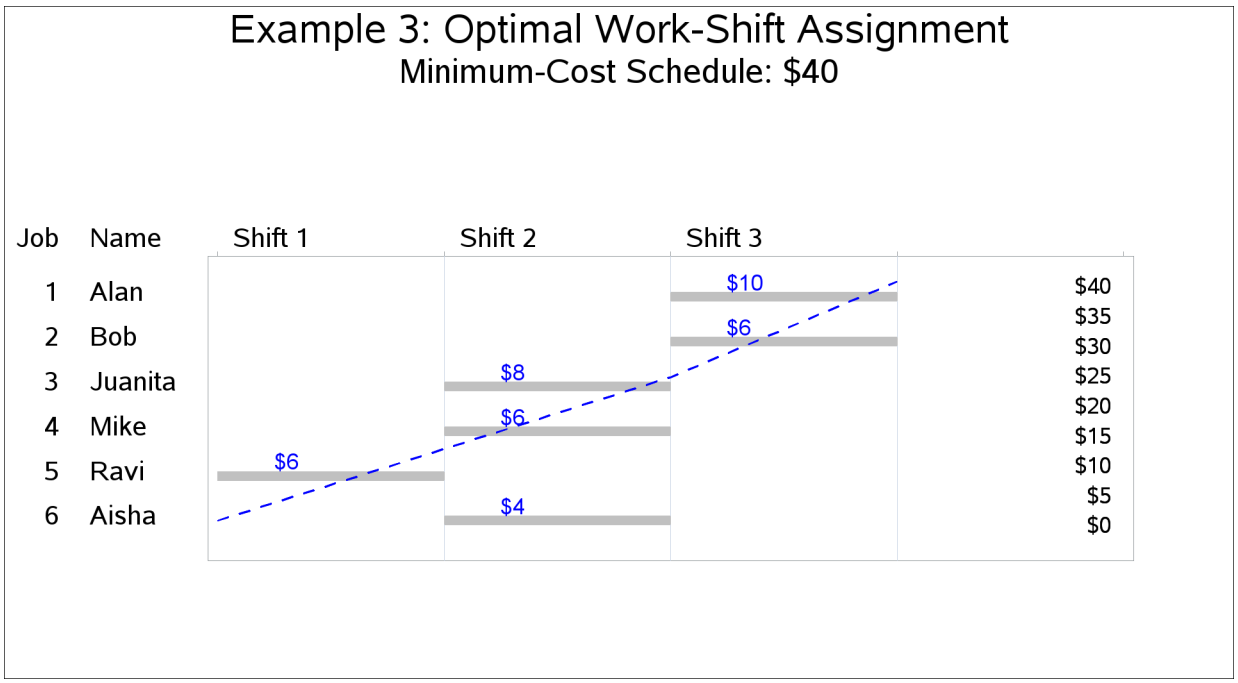
The cost of the optimal schedule, which corresponds to the solution shown in the following output, is $40.

**Solution to Optimal Work-Shift Scheduling Problem**

| Obs | W1 | W2 | W3 | W4 | W5 | W6 | C1 | C2 | C3 | C4 | C5 | C6 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 3 | 3 | 2 | 2 | 1 | 2 | 10 | 6 | 8 | 6 | 6 | 4 |

The minimum-cost schedule is displayed in the Gantt chart in Output 6.3.3.

**Output 6.3.3** Work-Shift Schedule with Minimum Cost




## Example 6.4: A Nonlinear Optimization Problem

This example illustrates how you can use the element constraint to represent almost any function between two variables in addition to representing nonstandard domains. Consider the following nonlinear optimization problem:

$$\text{maximize } f(x) = x_1^3 + 5x_2 - 2^{x_3}$$

$$\text{subject to } \begin{cases} x_1 - .5x_2 + x_3^2 \leq 50 \\ \text{mod}(x_1, 4) + .25x_2 \geq 1.5 \end{cases}$$

where $x_1$ is any integer in [–5, 5], $x_2$ is any *odd* integer in [–5, 9], and $x_3$ is any integer in [1, 10].

You can solve this problem by introducing four artificial variables, $y_1$–$y_4$, to represent each of the nonlinear terms. Let $y_1 = x_1^3$, $y_2 = 2^{x_3}$, $y_3 = x_3^2$, and $y_4 = \text{mod}(x_1, 4)$. You can represent the domains of $x_1$ and $x_2$ (which are not consecutive integers that start from 1) by using element constraints and index variables. For example, any of the following three element constraints specifies that the domain of $x_2$ is the set of odd integers in $[-5, 9]$:

```
con element(z2,-5 -3 -1 1 3 5 7 9,x2);
con element(z2, {ri in -5..9 by 2} ri, x2);
num range{ri in -5..9 by 2} = ri;
con element(z2,range,x2);
```

Any functional dependencies on $x_1$ or $x_2$ can now be defined using $z_1$ or $z_2$, respectively, as the index variable in an element constraint. Because the domain of $x_3$ is $[1, 10]$, you can directly use $x_3$ as the index variable in an element constraint to define dependencies on $x_3$.

For example, the following constraint specifies the function $y_1 = x_1^3$, $x_1 \in [-5, 5]$:

```
con element(z1,-125 -64 -27 -8 -1 0 1 8 27 64 125,y1);
/* or, con element(z1, {ri in -5..5} (ri**3), y1); */
num range{ri in -5..5} = ri**3;
con element(z1,range,y1);
```

To solve the problem, define the objective function as demonstrated in the following statements:

```
proc optmodel;
   set DOM{1..3} = [ (-5 .. 5) (-5 .. 9 by 2) (1 .. 10) ];
   var X{i in 1..3} integer >= min{j in DOM[i]} j <= max{j in DOM[i]} j;

   /* map the domain of X[1] and X[2] to 1 .. list size */
   var Z {1..2} integer;
   /* map nonlinear expressions */
   var Y {1..4} integer;

   /* Use an element constraint to represent noncontiguous domains */
   /* domains with negative numbers, and nonlinear functions. */
   /* Z[2] does not appear anywhere else. Its only purpose is
      to restrict X[2] to take a value from DOM[2]. */
   con MapDomainTo1ToCard{i in 1..2}:
      element(Z[i], {k in DOM[i]} k, X[i]);

   /* Functional Dependencies on X[1] */
   /* Y[1] = X[1]^3 -- Use Z[1] for X[1] for proper indexing */
   con Y1:
      element(Z[1], {k in DOM[1]} (k^3), Y[1]);

   /* Y[4] = mod(X[1], 4) */
   con Y4:
      element(Z[1], {k in DOM[1]} (mod(k,4)), Y[4]);


   /* Functional Dependencies on X[3] */
   /* Y[2] = 2^X[3] */
   con Y2:
      element(X[3], {k in DOM[3]} (2^k), Y[2]);

   /* Y[3] = X[3]^2 */
   con Y3:
      element(X[3], {k in DOM[3]} (k^2), Y[3]);

   /* X[1] - 0.5 * X[2] + X[3]^2 <= 50 */
   con Con1:
      X[1] - 0.5 * X[2] + Y[3] <= 50;

   /* mod(X[1],4) + 0.25 * X[2] >= 1.5 */
   con Con2:
      Y[4] + 0.25 * X[2] >= 1.5;

   /* Objective function: X[1]^3 + 5 * X[2] - 2^X[3] */
   max Objective = Y[1] + 5 * X[2] - Y[2];

   solve;
   print X Y Z;
quit;
```

Output 6.4.1 shows the solution that corresponds to the optimal objective value of 168.

**Output 6.4.1** Nonlinear Optimization Problem Solution

**The OPTMODEL Procedure**

| Problem Summary | |
|---|---|
| **Objective Sense** | Maximization |
| **Objective Function** | Objective |
| **Objective Type** | Linear |
| | |
| **Number of Variables** | 9 |
| **Bounded Above** | 0 |
| **Bounded Below** | 0 |
| **Bounded Below and Above** | 3 |
| **Free** | 6 |
| **Fixed** | 0 |
| **Binary** | 0 |
| **Integer** | 9 |
| | |
| **Number of Constraints** | 8 |
| **Linear LE (<=)** | 1 |
| **Linear EQ (=)** | 0 |
| **Linear GE (>=)** | 1 |
| **Linear Range** | 0 |
| **Alldiff** | 0 |
| **Element** | 6 |
| **GCC** | 0 |
| **Lexico (<=)** | 0 |
| **Lexico (<)** | 0 |
| **Pack** | 0 |
| **Reify** | 0 |
| | |
| **Constraint Coefficients** | 5 |

| Solution Summary | |
|---|---|
| **Solver** | CLP |
| **Variable Selection** | MINR |
| **Objective Function** | Objective |
| **Solution Status** | Optimal |
| **Objective Value** | 168 |
| | |
| **Solutions Found** | 1 |
| **Presolve Time** | 0.00 |
| **Solution Time** | 0.00 |

| [1] | X | Y | Z |
|---|---|---|---|
| **1** | 5 | 125 | 11 |
| **2** | 9 | 2 | 8 |
| **3** | 1 | 1 | |
| **4** | | 1 | |

## Example 6.5: Car Painting Problem

The car painting process is an important part of the automobile manufacturing industry. Purging (the act of changing colors during assembly) is expensive because of the added cost of wasted paint and solvents from each color change and the extra time that the purging process requires. The objective of the car painting problem is to sequence the cars in the assembly line in order to minimize the number of paint color changes (Sokol 2002; Trick 2004).

Suppose an assembly line contains 10 cars, which are ordered 1, 2, . . . , 10. A car must be painted within three positions of its assembly order. For example, car 5 can be painted in positions 2 through 8. Cars 1, 5, and 9 are red; 2, 6, and 10 are blue; 3 and 7 green; and 4 and 8 are yellow. The initial sequence 1, 2, . . . , 10 corresponds to the color pattern RBGYRBGYRB and has nine purgings. The objective is to find a solution that minimizes the number of purgings.

This problem can be formulated as a CSP as follows:

- The input is the color of each car currently on the assembly line.
- The output variables are the slot in which each car will be painted and whether purging will be required after that painting operation.
- Set the bounds of the slot variables to their feasible range, at most three slots before or after the car's current position.
- To determine whether purging is needed, use another variable to store the color of the car painted in each slot.
- Use an element constraint to determine the color used in each slot from the car assigned to that slot.
- Use a reify constraint to determine whether two consecutive slots are assigned different colors, and thus whether purging is required.
- Finally, use a linear constraint to limit the total number of purgings.

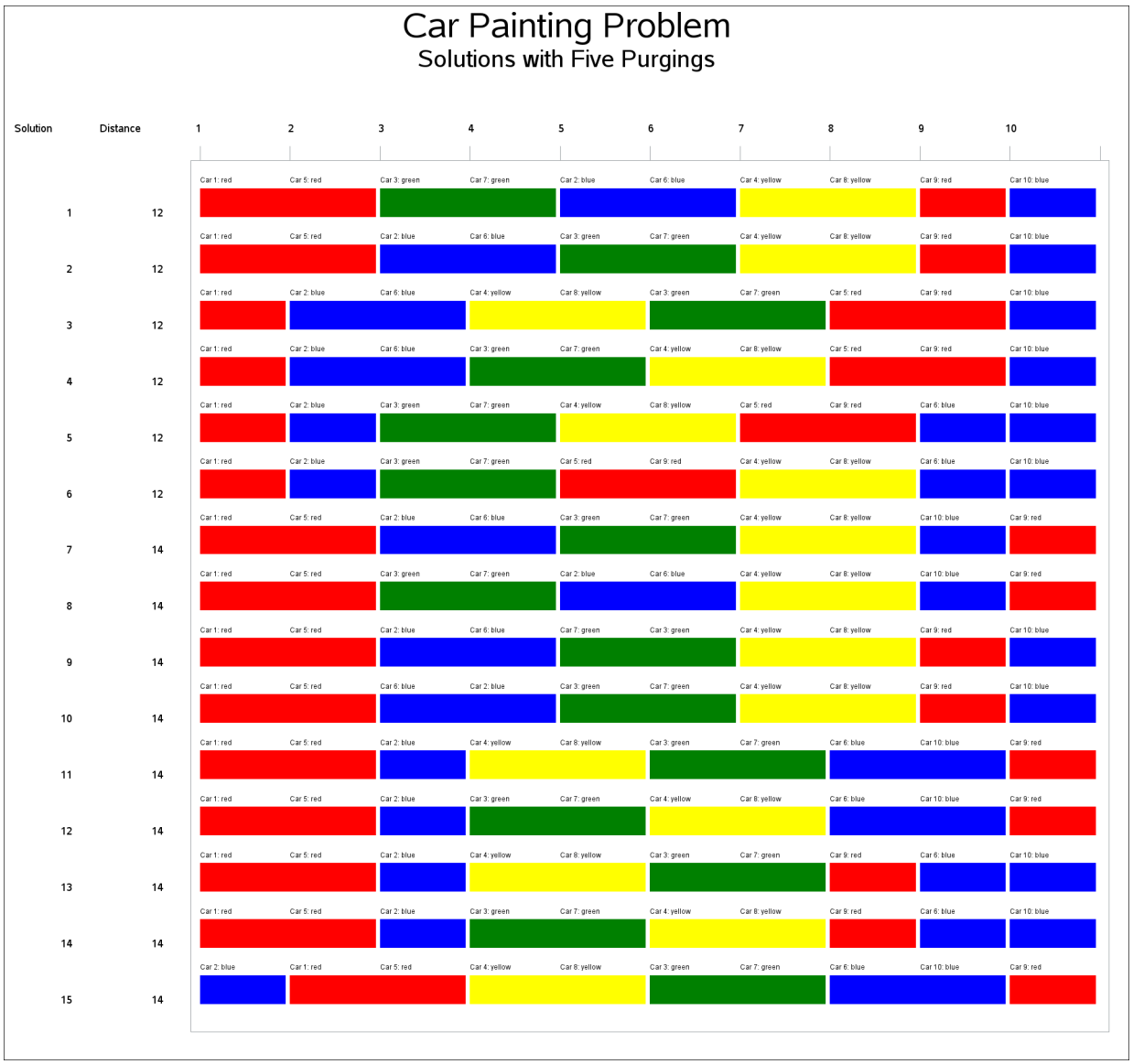
The following %CAR_PAINTING macro determines all feasible solutions for the number of purgings that is specified as a parameter to the macro:

```
%macro car_painting(purgings);
   proc optmodel;
      num nCars = 10;
      /* a car is identified by its original slots */
      set SLOTS = 1..nCars;

      /* maximum reshuffling of any car on the line*/
      num maxMove init 3;
      /* which car is in slot i. */
      var S {si in SLOTS} integer >= max(1, si - maxMove)
                                  <= min(nCars, si + maxMove) ;

      /* which color the car in slot i is. */
      /* Red=1; Blue=2; Green=3; Yellow=4 */
      num nColors=4;
```

```
      num colorOf{SLOTS} = [ 1 2 3 4 1 2 3 4 1 2 ];
      var C {SLOTS} integer >= 1 <= nColors;

      con ElementCon {i in SLOTS}:
         element(S[i], colorOf, C[i]);

      /* A car can be painted only once. */
      con PaintOnlyOnce:
         alldiff(S);

      /* Whether there is a purge after slot i.
        You can ignore any purging that would happen at the end of the shift. */
      var P {SLOTS diff {nCars}} binary;

      con ReifyCon {i in SLOTS diff {nCars}}:
         reify(P[i], C[i] ne C[i+1]);

      /* Calculate the number of purgings. */
      con PurgingsCon:
         sum {i in SLOTS diff {nCars}} P[i] <= &purgings;

      solve with CLP / findall;
      /* Replicate typical PROC CLP output from PROC OPTMODEL arrays */
      create data car_ds(drop=k) from [k]=(1.._NSOL_)
         {i in SLOTS} <col('S'||i)=S[i].sol[k]>
         {i in SLOTS} <col('C'||i)=C[i].sol[k]>;
   quit;
%mend;
%car_painting(5);
```

The problem is infeasible for four purgings. The CLP solver finds 87 possible solutions for the five-purgings problem. The solutions are sorted by the total distance that all cars are moved in the sequencing, which ranges from 12 to 22 slots. The first 15 solutions are displayed in the Gantt chart in Output 6.5.1. Each row represents a solution, and each color transition represents a paint purging.

**Output 6.5.1** Car Painting Schedule with Five Purgings




## Example 6.6: Scene Allocation Problem

The scene allocation problem consists of deciding when to shoot each scene of a movie in order to minimize the total production cost (Van Hentenryck 2002). Each scene involves a number of actors, and at most five scenes a day can be shot. All actors who appear in a scene must be present in the studio on the day the scene is shot. Each actor earns a daily rate for each day spent in the studio, regardless of the number of scenes in which he or she appears on that day. The goal is to shoot the movie for the lowest possible production cost.

The actors' names, their daily fees, and the scenes in which they appear are contained in the Scene data set, which is shown in Output 6.6.1. The data set variables S_Var1, . . . , S_Var9 indicate the scenes in which the

actor appears. For example, the first observation indicates that Patt's daily fee is 26,481 and that Patt appears in scenes 2, 5, 7, 10, 11, 13, 15, and 17.

**Output 6.6.1** Scene Data Set

| Obs | Number | Actor | DailyFee | S_Var1 | S_Var2 | S_Var3 | S_Var4 | S_Var5 | S_Var6 | S_Var7 | S_Var8 | S_Var9 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **1** | 1 | Patt | 26481 | 2 | 5 | 7 | 10 | 11 | 13 | 15 | 17 | . |
| **2** | 2 | Casta | 25043 | 4 | 7 | 9 | 10 | 13 | 16 | 19 | . | . |
| **3** | 3 | Scolaro | 30310 | 3 | 6 | 9 | 10 | 14 | 16 | 17 | 18 | . |
| **4** | 4 | Murphy | 4085 | 2 | 8 | 12 | 13 | 15 | . | . | . | . |
| **5** | 5 | Brown | 7562 | 2 | 3 | 12 | 17 | . | . | . | . | . |
| **6** | 6 | Hacket | 9381 | 1 | 2 | 12 | 13 | 18 | . | . | . | . |
| **7** | 7 | Anderson | 8770 | 5 | 6 | 14 | . | . | . | . | . | . |
| **8** | 8 | McDougal | 5788 | 3 | 5 | 6 | 9 | 10 | 12 | 15 | 16 | 18 |
| **9** | 9 | Mercer | 7423 | 3 | 4 | 5 | 8 | 9 | 16 | . | . | . |
| **10** | 10 | Spring | 3303 | 5 | 6 | . | . | . | . | . | . | . |
| **11** | 11 | Thompson | 9593 | 6 | 9 | 12 | 15 | 18 | . | . | . | . |

There are 19 scenes. At most 5 scenes can be filmed in one day, so at least four days are needed to schedule all the scenes ($\lceil \frac{19}{5} \rceil = 4$). Let $S_{jk}$ be a binary variable that equals 1 if scene $j$ is shot on day $k$. Let $A_{ik}$ be another binary variable that equals 1 if actor $i$ is present on day $k$. The input daily_fee$_i$ is the daily cost of the $i$th actor.

The objective function that represents the total production cost is

$$\min \sum_{i=1}^{11} \sum_{k=1}^{4} \text{daily\_fee}_i \times A_{ik}$$

This example illustrates the use of symmetry-breaking constraints. In this model, the "1" in day 1 does not refer to sequence but simply to the label of the day. Thus, you can call day 1 the day on which scene 1 is shot, whichever day that is. Similarly, either scene 2 is shot on the same day as scene 1 (day 1) or it is shot on another day, which you can call day 2. Scene 3 is shot either on one of those two days or on another day. Adding constraints that eliminate symmetry can significantly improve the performance of a CLP model. In this model, the symmetry-breaking constraints prevent the solver from considering three other assignments that do not differ in any meaningful way.

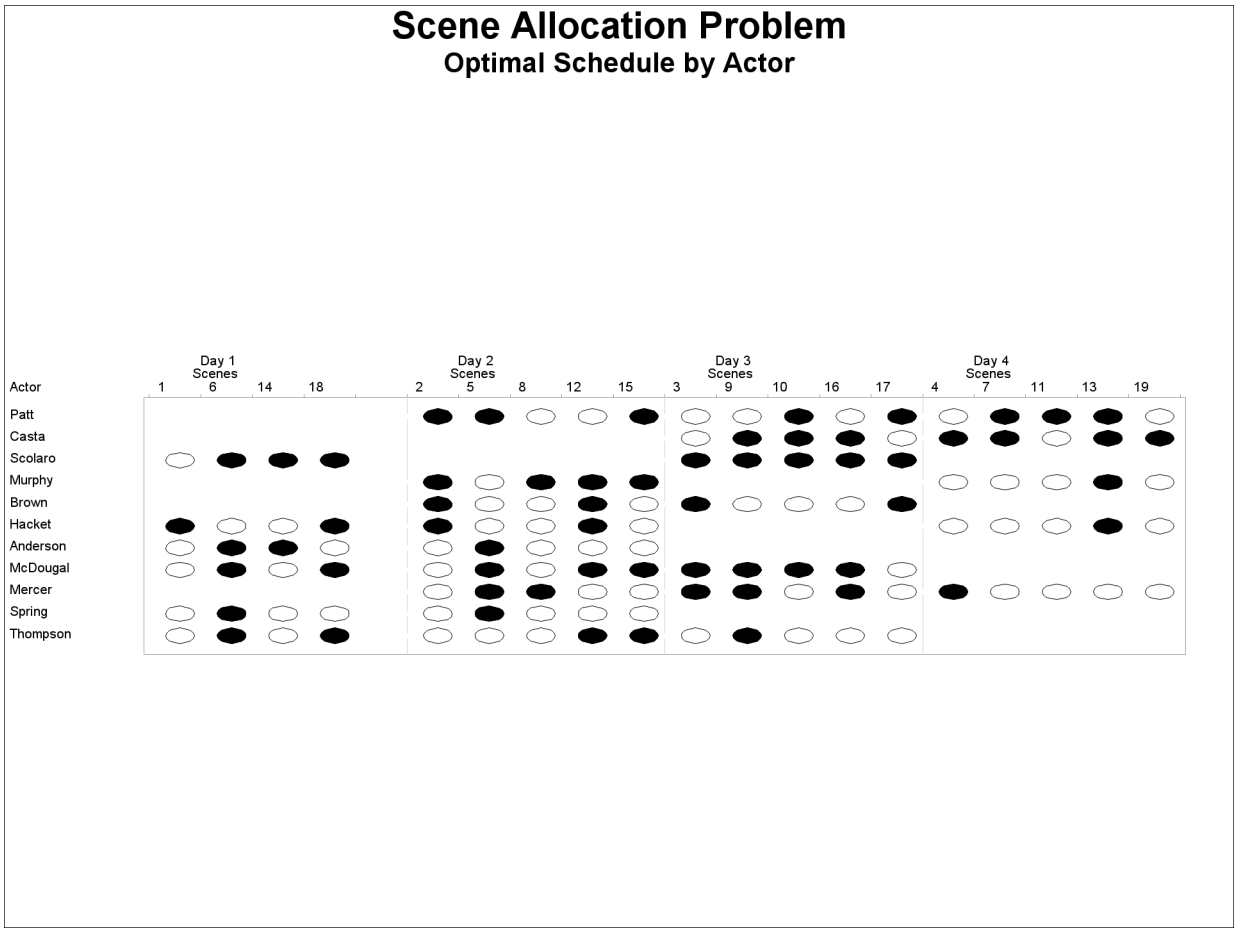
The following PROC OPTMODEL statements implement these ideas:

```
proc optmodel;
   set ACTORS;
   str actor_name {ACTORS};
   num daily_fee {ACTORS};
   num most_scenes = 9; /* most scenes by any actor */
   num scene_list {ACTORS, 1..most_scenes};
   read data scene into ACTORS=[_N_]
      actor_name=Actor daily_fee=DailyFee
      {j in 1..most_scenes} <scene_list[_N_,j]=col('S_Var'||j)>;
   print actor_name daily_fee scene_list;

   set SCENES_actor {i in ACTORS} =
      (setof {j in 1..most_scenes} scene_list[i,j]) diff {.};
```

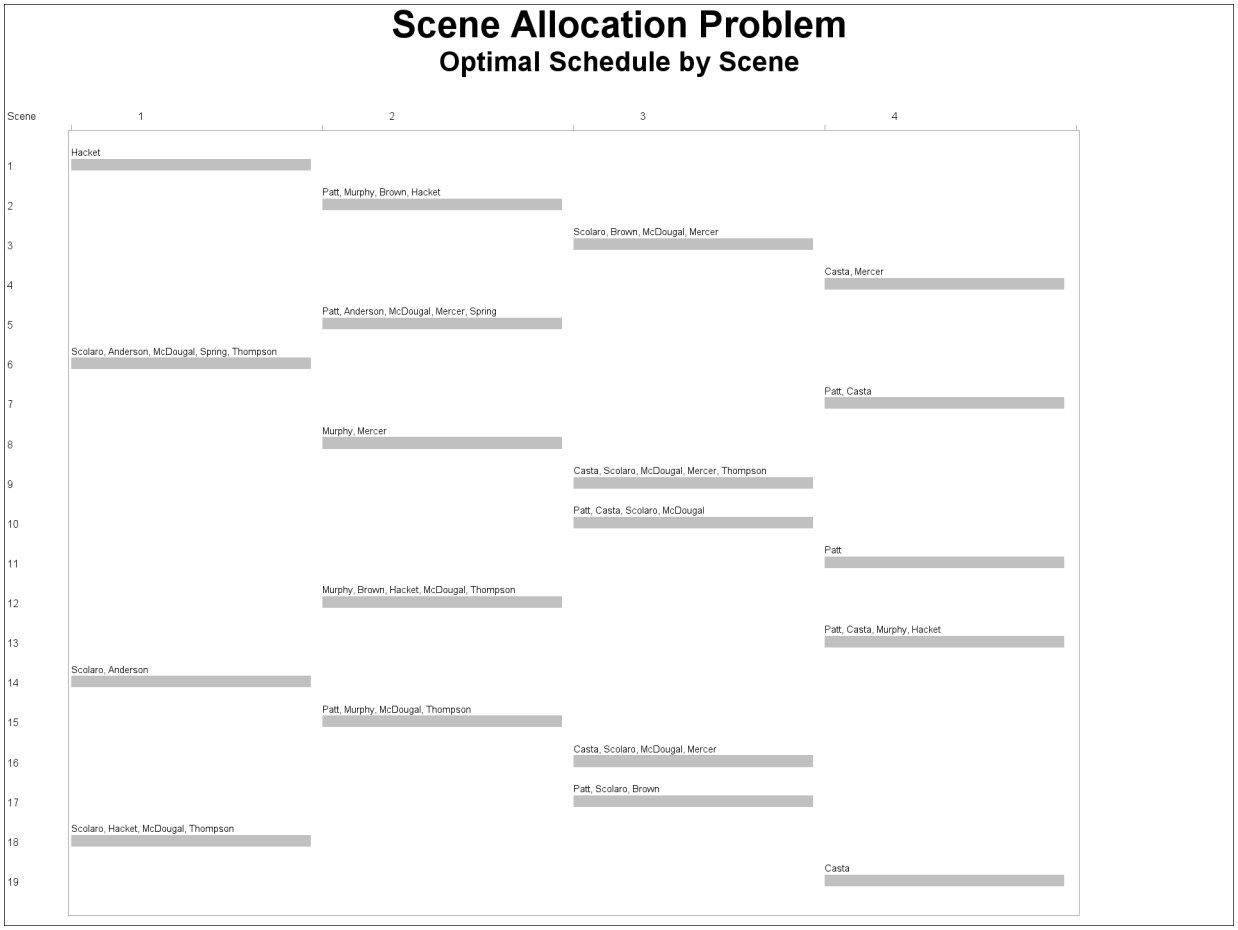
```
   set SCENES = 1..19;
   set DAYS = 1..4;

   /* Indicates if actor i is present on day k. */
   var A {ACTORS, DAYS} binary;

   /* Indicates if scene j is shot on day k. */
   var S {SCENES, DAYS} binary;

   /* Every scene is shot exactly once.*/
   con SceneCon {j in SCENES}:
      gcc({k in DAYS} S[j,k], {<1,1,1>});

   /* At least 4 and at most 5 scenes are shot per day. */
   con NumScenesPerDayCon {k in DAYS}:
      gcc({j in SCENES} S[j,k], {<1,4,5>});

   /* Actors for a scene must be present on day of shooting. */
   con LinkCon {i in ACTORS, j in SCENES_actor[i], k in DAYS}:
      S[j,k] <= A[i,k];

   /* Symmetry-breaking constraints. Without loss of any generality, you
      can assume Scene1 to be shot on day 1, Scene2 to be shot on day 1
      or day 2, and Scene3 to be shot on either day 1, day 2, or day 3.  */
   fix S[1,1] = 1;
   for {k in 2..4} fix S[1,k] = 0;
   for {k in 3..4} fix S[2,k] = 0;
   fix S[3,4] = 0;

   /* If Scene2 is shot on day 1, (as opposed to day 2) */
   /* then Scene3 can be shot on day 1 or day 2 (but not day 3). */
   con Symmetry:
      S[2,1] + S[3,3] <= 1;

   /* Minimize total cost. */
   min TotalCost = sum {i in ACTORS, k in DAYS} daily_fee[i] * A[i,k];

   /* Set lower and upper bounds for the objective value   */
   /* Lower bound: every actor appears on one day.         */
   /* Upper bound: every actor appears on all four days.   */
   num obj_lb = sum {i in ACTORS} daily_fee[i];
   num obj_ub = sum {i in ACTORS, k in DAYS} daily_fee[i];
   put obj_lb= obj_ub=;
   con TotalCost_bounds:
      obj_lb <= TotalCost <= obj_ub;

   solve with CLP / varselect=maxc;
quit;
```

The optimal production cost is 334,144, as reported in the _OROPTMODEL_ macro variable. The corresponding actor schedules and scene schedules are displayed in Output 6.6.2 and Output 6.6.3, respectively.

**Output 6.6.2** Scene Allocation Problem: Actor Schedules



**Output 6.6.3** Scene Allocation Problem: Scene Schedules




## Example 6.7: Car Sequencing Problem

This problem is an instance of a category of problems known as the car sequencing problem. A considerable amount of literature discusses this problem (Dincbas, Simonis, and Van Hentenryck 1988; Gravel, Gagne, and Price 2005; Solnon et al. 2008).

A number of cars are to be produced on an assembly line. Each car is customized to include a specific set of options, such as air conditioning, a sunroof, a navigation system, and so on. The assembly line moves through several workstations for installation of these options. The cars cannot be positioned randomly, because each workstation has limited capacity and needs time to set itself up to install the options as the car moves in front of the station. These capacity constraints are formalized using constraints of the form $m$ out of $N$, which indicates that the workstation can install the option on $m$ out of every sequence of $N$ cars. The car sequencing problem is to determine a sequencing of the cars on the assembly line that satisfies the demand constraints for each set of car options and the capacity constraints for each workstation.

This example comes from Dincbas, Simonis, and Van Hentenryck (1988). Ten cars need to be customized with five possible options. A class of car is defined by a specific set of options; there are six classes of cars.

The data are presented in Table 6.6.

**Table 6.6** Option Installation Data

| Option | | Capacity | Car Class | | | | | |
|---|---|---|---|---|---|---|---|---|
| **Name** | **Type** | **m/N** | **1** | **2** | **3** | **4** | **5** | **6** |
| Option 1 | 1 | 1/2 | 1 | 0 | 0 | 0 | 1 | 1 |
| Option 2 | 2 | 2/3 | 0 | 0 | 1 | 1 | 0 | 1 |
| Option 3 | 3 | 1/3 | 1 | 0 | 0 | 0 | 1 | 0 |
| Option 4 | 4 | 2/5 | 1 | 1 | 0 | 1 | 0 | 0 |
| Option 5 | 5 | 1/5 | 0 | 0 | 1 | 0 | 0 | 0 |
| **Number of Cars** | | | **1** | **1** | **2** | **2** | **2** | **2** |

For example, car class 4 requires installation of option 2 and option 4, and two cars of this class are required. The workstation for option 2 can process only two out of every sequence of three cars. The workstation for option 4 has even less capacity—two out of every five cars.

The data for this problem are used to create a SAS data set, which drives the generation of variables and constraints in PROC OPTMODEL.

The decision variables for this problem are shown in Table 6.7.

**Table 6.7** Decision Variables

| Variable Definition | Description |
|---|---|
| S{SLOTS} >=1 <= 6 | $S_i$ is the class of cars assigned to slot $i$. |
| var O {SLOTS, OPTIONS} binary | $O_{ij} = 1$ if the class assigned to slot $i$ needs option $j$. |

The following SAS statements express the workstation capacity constraints by using a set of linear constraints for each workstation. A single GCC constraint expresses the demand constraints for each car class. An element constraint for each option variable expresses the relationships between slot variables and option variables.

This model also includes a set of redundant constraints, in the sense that the preceding logical constraints correctly represent the set of feasible solutions. However, the redundant constraints provide the solver with further information specific to this problem, significantly improving the efficiency of domain propagation. Redundant constraints are a core fixture of CLP models. They can determine whether a model will linger and be unsolvable for real data or will produce instant results.

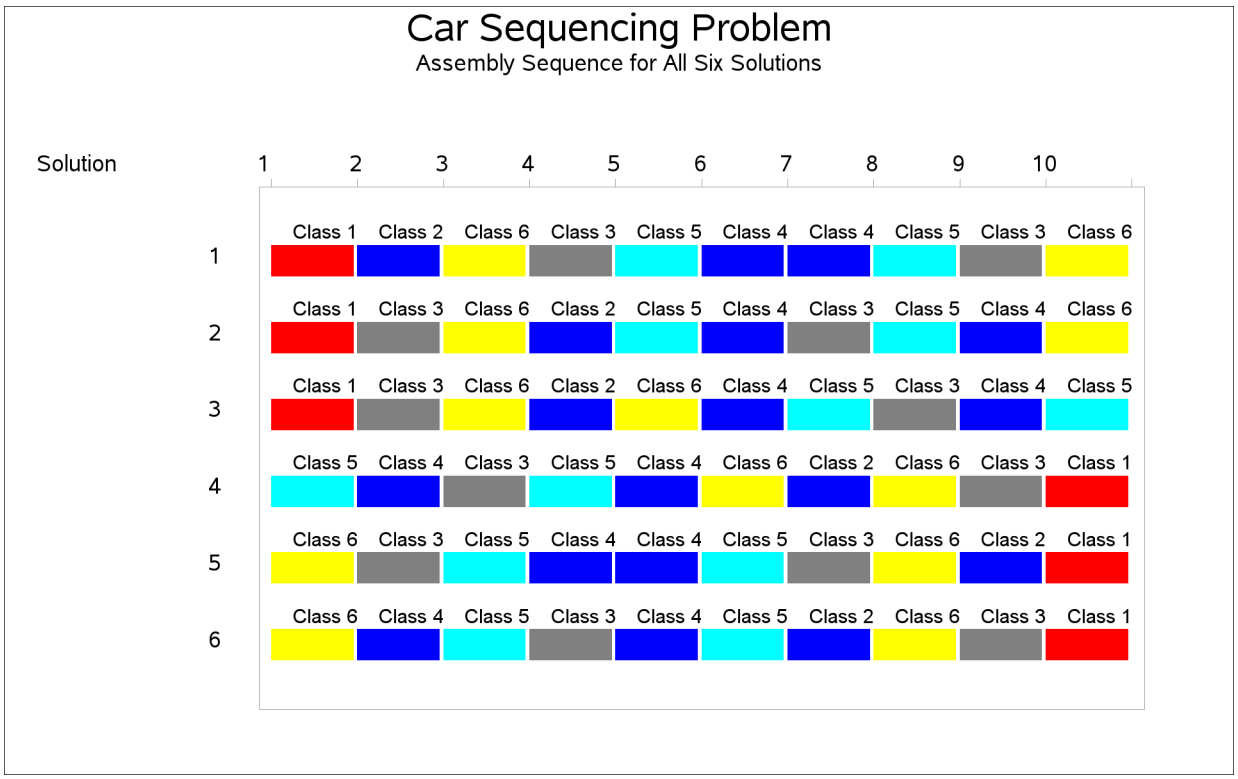
The idea behind the redundant constraint in this model is the following realization: if the workstation for option $j$ has capacity $r$ out of $s$, then at most $r$ cars in the sequence $(n-s+1),\ldots,n$ can have option $j$, where $n$ is the total number of cars. Consequently, at least $n_j - r$ cars in the sequence $1,\ldots,n-s$ must have option $j$, where $n_j$ is the number of cars that have option $j$. Generalizing this further, at least $n_j - k \times r$ cars in the sequence $1,\ldots,(n-k\times s)$ must have option $j$, $k = 1,\ldots,\lfloor n/s \rfloor$.

```
data class_data;
   input class cars_cls;
   datalines;
1 1
2 1
3 2
4 2
5 2
6 2
;

data option_data;
   input option max blSz class1-class6;
   datalines;
1 1 2 1 0 0 0 1 1
2 2 3 0 0 1 1 0 1
3 1 3 1 0 0 0 1 0
4 2 5 1 1 0 1 0 0
5 1 5 0 0 1 0 0 0
;

%macro car_sequencing(outdata);
   proc optmodel;
      set CLASSES;
      num nClasses = card(CLASSES);
      num cars_cls {CLASSES};
      read data class_data into CLASSES=[class] cars_cls;

      set OPTIONS;
      num max {OPTIONS};
      num blSz {OPTIONS};
      num list {OPTIONS, CLASSES};
      num cars_opt {i in OPTIONS} = sum {k in CLASSES} cars_cls[k] * list[i,k];
      read data option_data into OPTIONS=[option] max blSz
         {k in CLASSES} <list[option,k]=col('class'||k)>;

      num nCars = sum {k in CLASSES} cars_cls[k];
      set SLOTS = 1..nCars;

      /* Declare Variables */
      /* Slot variables: S[i] - class of car assigned to Slot i */
      var S {SLOTS} integer >= 1 <= nClasses;

      /* Option variables: O[i,j]
      - indicates if class assigned to Slot i needs Option j */
      var O {SLOTS, OPTIONS} binary;

      /* Capacity Constraints: for each option j */
      /* Install in at most max[j] out of every sequence of blSz[j] cars */
      con CapacityCon {j in OPTIONS, i in 0..(nCars-blSz[j])}:
         sum {k in 1..blSz[j]} O[i+k,j] <= max[j];

      /* Demand Constraints: for each class k */
```

```
      /* Exactly cars_cls[k] cars */
      con MeetDemandCon:
         gcc(S, setof{k in CLASSES} <k,cars_cls[k],cars_cls[k]>);

      /* Element Constraints: For each slot i and each option j */
      /* relate the slot variable to the option variables.      */
      /* O[i,j] = list[j,S[i]]                                  */
      con OptionsAtSlotCon {i in SLOTS, j in OPTIONS}:
         element(S[i], {k in CLASSES} list[j,k], O[i,j]);

      /* Redundant Constraints to improve efficiency - for every  */
      /*    option j.                                             */
      /* At most max[j] out of every sequence of blSz[j] cars     */
      /*    requires option j.                                    */
      /* All the other slots contain at least cars_opt[j] - max[j]*/
      /*    cars with option j                                    */
      con BoundRemainingCon {j in OPTIONS, i in 1..(nCars/blSz[j])}:
         sum {k in 1..(nCars-i*blSz[j])} O[k,j] >= cars_opt[j] - i * max[j];

      solve with CLP / varselect=minrmaxc findall;

      /* Replicate typical PROC CLP output from PROC OPTMODEL arrays */
      create data &outdata.(drop=sol) from [sol]=(1.._NSOL_)
         {i in SLOTS} <col('S_'||i)=S[i].sol[sol]>
         {i in SLOTS, j in OPTIONS} <col('O_'||i||'_'||j)=O[i,j].sol[sol]>;
   quit;
%mend;
%car_sequencing(sequence_out);
```

This problem has six solutions, as shown in Output 6.7.1.

**Output 6.7.1** Car Sequencing




## Example 6.8: Balanced Incomplete Block Design

Balanced incomplete block design (BIBD) generation is a standard combinatorial problem from design theory. The concept was originally developed in the design of statistical experiments; applications have expanded to other fields, such as coding theory, network reliability, and cryptography. A BIBD is an arrangement of $v$ distinct objects into $b$ blocks such that the following conditions are met:

- Each block contains exactly $k$ distinct objects.
- Each object occurs in exactly $r$ different blocks.
- Every two distinct objects occur together in exactly $\lambda$ blocks.

A BIBD is therefore specified by its parameters $(v, b, r, k, \lambda)$. It can be proved that when a BIBD exists, its parameters must satisfy the following conditions:

- $rv = bk$
- $\lambda(v-1) = r(k-1)$
- $b \geq v$

The preceding conditions are not sufficient to guarantee the existence of a BIBD (Prestwich 2001). For example, the parameters $(15, 21, 7, 5, 2)$ satisfy the preceding conditions, but a BIBD that has these parameters does not exist. Computational methods of BIBD generation usually suffer from combinatorial explosion, in part because of the large number of symmetries: for any solution, any two objects or blocks can be exchanged to obtain another solution.

This example demonstrates how to express a BIBD problem as a CSP and how to use lexicographic ordering constraints to break symmetries. Note that this example is for illustration only. SAS provides an autocall macro, MKTBIBD, for solving BIBD problems. The most direct CSP model for BIBD, as described in Meseguer and Torras (2001), represents a BIBD as a $v \times b$ matrix $\boldsymbol{X}$. Each matrix entry is a Boolean decision variable $X_{i,c}$ that satisfies $X_{i,c} = 1$ if and only if block $c$ contains object $i$. The condition that each object occurs in exactly $r$ blocks (or, equivalently, that there are $r$ 1s per row) can be expressed as $v$ linear constraints:

$$\sum_{c=1}^{b} X_{i,c} = r \quad \text{for } i = 1, \ldots, v$$

Alternatively, you can use global cardinality constraints to ensure that there are exactly $b - r$ 0s and $r$ 1s in $X_{i,1}, \ldots, X_{i,b}$ for each object $i$:

$$\text{gcc}(X_{i,1}, \ldots, X_{i,b}) = ((0, 0, b - r)(1, 0, r)) \quad \text{for } i = 1, \ldots, v$$

Similarly, you can use the following constraints to specify the condition that each block contain exactly $k$ objects (there are $k$ 1s per column):

$$\text{gcc}(X_{1,c}, \ldots, X_{v,c}) = ((0, 0, v - k)(1, 0, k)) \quad \text{for } c = 1, \ldots, b$$

To enforce the final condition that every two distinct objects occur together in exactly $\lambda$ blocks (equivalently, that the scalar product of every pair of rows equal $\lambda$), you can introduce the auxiliary variables $P_{i,j,c}$ for every $i < j$, which indicate whether objects $i$ and $j$ both occur in block $c$. The following reify constraint ensures that $P_{i,j,c} = 1$ if and only if block $c$ contains both objects $i$ and $j$:

$$\text{reify } P_{i,j,c} : (X_{i,c} + X_{j,c} = 2)$$

The following constraints ensure that the final condition holds:

$$\text{gcc}(P_{i,j,1}, \ldots, P_{i,j,b}) = ((0, 0, b - \lambda)(1, 0, \lambda)) \quad \text{for } i = 1, \ldots, v - 1 \text{ and } j = i + 1, \ldots, v$$

The objects and the blocks are interchangeable, so the matrix $\boldsymbol{X}$ has total row symmetry and total column symmetry. Because of the constraints on the rows, no pair of rows can be equal unless $r = \lambda$. To break the row symmetry, you can impose strict lexicographic ordering on the rows of $X$ as follows:

$$(X_{i,1}, \ldots, X_{i,b}) <_{\text{lex}} (X_{i-1,1}, \ldots, X_{i-1,b}) \quad \text{for } i = 2, \ldots, v$$

To break the column symmetry, you can impose lexicographic ordering on the columns of $X$ as follows:

$$(X_{1,c}, \ldots, X_{v,c}) \leq_{\text{lex}} (X_{1,c-1}, \ldots, X_{v,c-1}) \quad \text{for } c = 2, \ldots, b$$

The following SAS macro incorporates all the preceding constraints. For the specified parameters $(v, b, r, k, \lambda)$, the macro either finds BIBDs or proves that a BIBD does not exist.

```
%macro bibd(v, b, r, k, lambda, out=bibdout);
   /* Arrange v objects into b blocks such that:
         (i) each object occurs in exactly r blocks,
         (ii) each block contains exactly k objects,
         (iii) every pair of objects occur together in exactly lambda blocks.

      Equivalently, create a binary matrix with v rows and b columns,
      with r 1s per row, k 1s per column,
      and scalar product lambda between any pair of distinct rows.
   */

   /* Check necessary conditions */
   %if (%eval(&r * &v) ne %eval(&b * &k)) or
      (%eval(&lambda * (&v - 1)) ne %eval(&r * (&k - 1))) or
      (&v > &b) %then %do;
      %put BIBD necessary conditions are not met.;
      %goto EXIT;
   %end;

   proc optmodel;
      num v = &v;
      num b = &b;
      num r = &r;
      num k = &k;
      num lambda = &lambda;
      set OBJECTS = 1..v;
      set BLOCKS = 1..b;

      /* Decision variable X[i,c] = 1 iff object i occurs in block c. */
      var X {OBJECTS, BLOCKS} binary;

      /* Mandatory constraints: */
      /* (i) Each object occurs in exactly r blocks. */
      con Exactly_r_blocks {i in OBJECTS}:
         gcc({c in BLOCKS} X[i,c], {<0,0,b-r>,<1,0,r>});

      /* (ii) Each block contains exactly k objects. */
      con Exactly_k_objects {c in BLOCKS}:
         gcc({i in OBJECTS} X[i,c], {<0,0,v-k>,<1,0,k>});

      /* (iii) Every pair of objects occurs in exactly lambda blocks. */
      set PAIRS = {i in OBJECTS, j in OBJECTS: i < j};
      /* auxiliary variable P[i,j,c] = 1 iff both i and j occur in c */
      var P {PAIRS, BLOCKS} binary;
      con Pairs_reify {<i,j> in PAIRS, c in BLOCKS}:
         reify(P[i,j,c], X[i,c] + X[j,c] = 2);
      con Pairs_gcc {<i,j> in PAIRS}:
         gcc({c in BLOCKS} P[i,j,c], {<0,0,b-lambda>,<1,0,lambda>});

      /* symmetry-breaking constraints: */
      /* Break row symmetry via lexicographic ordering constraints. */
      con Symmetry_i {i in OBJECTS diff {1}}:
         lexico({c in BLOCKS} X[i,c] < {c in BLOCKS} X[i-1,c]);
```

```
      /* Break column symmetry via lexicographic ordering constraints. */
      con Symmetry_c {c in BLOCKS diff {1}}:
         lexico({i in OBJECTS} X[i,c] <= {i in OBJECTS} X[i,c-1]);

      solve with CLP / varselect=FIFO;
      create data &out from
         {i in OBJECTS, c in BLOCKS} <col('X'||i||'_'||c)=X[i,c]>;
   quit;
   %put &_oroptmodel_;
%EXIT:
%mend bibd;
```

The following statement invokes the macro to find a BIBD design for the parameters $(15, 15, 7, 7, 3)$:

```
%bibd(15,15,7,7,3);
```

The output is displayed in Output 6.8.1.

**Output 6.8.1** Balanced Incomplete Block Design for (15,15,7,7,3)

**Balanced Incomplete Block Design Problem**
**(15, 15, 7, 7, 3)**

| Obs | Block1 | Block2 | Block3 | Block4 | Block5 | Block6 | Block7 | Block8 | Block9 | Block10 | Block11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **1** | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| **2** | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| **3** | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| **4** | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 |
| **5** | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 1 |
| **6** | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 1 |
| **7** | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 |
| **8** | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 |
| **9** | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 |
| **10** | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 |
| **11** | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 |
| **12** | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 1 |
| **13** | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 0 |
| **14** | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 |
| **15** | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 |

| Obs | Block12 | Block13 | Block14 | Block15 |
|---|---|---|---|---|
| **1** | 0 | 0 | 0 | 0 |
| **2** | 0 | 0 | 0 | 0 |
| **3** | 1 | 1 | 1 | 0 |
| **4** | 1 | 1 | 0 | 1 |
| **5** | 1 | 0 | 0 | 1 |
| **6** | 0 | 1 | 1 | 0 |
| **7** | 0 | 0 | 1 | 1 |
| **8** | 1 | 0 | 1 | 1 |
| **9** | 0 | 1 | 0 | 1 |
| **10** | 0 | 1 | 0 | 1 |
| **11** | 1 | 0 | 1 | 0 |
| **12** | 0 | 0 | 1 | 1 |
| **13** | 0 | 1 | 1 | 0 |
| **14** | 1 | 1 | 0 | 0 |
| **15** | 1 | 0 | 0 | 0 |

## Example 6.9: Progressive Party Problem

This example demonstrates the use of the pack constraint to solve an instance of the progressive party problem (Smith et al. 1996). In the original progressive party problem, a number of yacht crews and their boats congregate at a yachting rally. In order for each crew to socialize with as many other crews as possible, some of the boats are selected to serve as "host boats" for six rounds of parties. The crews of the host boats stay with their boats for all six rounds. The crews of the remaining boats, called "guest crews," are assigned to visit a different host boat in each round.

Given the number of boats at the rally, the capacity of each boat, and the size of each crew, the objective of the original problem is to assign all the guest crews to host boats for each of the six rounds, using the minimum number of host boats. The partitioning of crews into guests and hosts is fixed throughout all rounds. No two crews should meet more than once. The assignments are constrained by the spare capacities (total capacity minus crew size) of the host boats and the crew sizes of the guest boats. Some boats cannot be hosts (zero spare capacity), and other boats must be hosts.

In this instance of the problem, the designation of the minimum requirement of 13 hosts is assumed (boats 1 through 12 and 14). The formulation solves up to eight rounds, but only two rounds are scheduled for this example. The total capacities and crew sizes of the boats are shown in Figure 6.3.

**Figure 6.3** Progressive Party Problem Input

**Progressive Party Problem Input**

| boatnum | capacity | crewsize |
|---|---|---|
| 1 | 6 | 2 |
| 2 | 8 | 2 |
| 3 | 12 | 2 |
| 4 | 12 | 2 |
| 5 | 12 | 4 |
| 6 | 12 | 4 |
| 7 | 12 | 4 |
| 8 | 10 | 1 |
| 9 | 10 | 2 |
| 10 | 10 | 2 |
| 11 | 10 | 2 |
| 12 | 10 | 3 |
| 13 | 8 | 4 |
| 14 | 8 | 2 |
| 15 | 8 | 3 |
| 16 | 12 | 6 |
| 17 | 8 | 2 |
| 18 | 8 | 2 |
| 19 | 8 | 4 |
| 20 | 8 | 2 |
| 21 | 8 | 4 |

| boatnum | capacity | crewsize |
|---|---|---|
| 22 | 8 | 5 |
| 23 | 7 | 4 |
| 24 | 7 | 4 |
| 25 | 7 | 2 |
| 26 | 7 | 2 |
| 27 | 7 | 4 |
| 28 | 7 | 5 |
| 29 | 6 | 2 |
| 30 | 6 | 4 |
| 31 | 6 | 2 |
| 32 | 6 | 2 |
| 33 | 6 | 2 |
| 34 | 6 | 2 |
| 35 | 6 | 2 |
| 36 | 6 | 2 |
| 37 | 6 | 4 |
| 38 | 6 | 5 |
| 39 | 9 | 7 |
| 40 | 0 | 2 |
| 41 | 0 | 3 |
| 42 | 0 | 4 |

The following statements and DATA steps process the data and designate host boats:

```
data hostability;
   set capacities;
   spareCapacity = capacity - crewsize;
run;

data hosts guests;
   set hostability;
   if (boatnum <= 12 or boatnum eq 14) then do;
      output hosts;
   end;
   else do;
      output guests;
   end;
run;

/* sort so guest boats with larger crews appear first */
proc sort data=guests;
   by descending crewsize;
run;

data capacities;
   format boatnum capacity 2.;
   set hosts guests;
   seqno = _n_;
run;
```

To model the progressive party problem for the CLP solver, first define the following sets of variables:

- Item variables $x_{it}$ contain the host boat number for the assignment of guest boat $i$ in round $t$.
- Load variables $L_{ht}$ contain the load of host boat $h$ in round $t$.
- Variable $m_{ijt}$ are binary variables that take a value of 1 if and only if guest boats $i$ and $j$ are assigned to the same host boat in round $t$.

Next, describe the set of constraints that are used in the model:

- All-different constraints ensure that a guest boat is not assigned to the same host boat in different rounds.
- Reify constraints regulate the values that are assigned to the aforementioned indicator variables $m_{ijt}$.
- The reified indicator variables appear in linear constraints to enforce the requirement to meet no more than once.
- One pack constraint per round maintains the capacity limits of the host boats.
- Finally, a symmetry-breaking linear constraint orders the host boat assignments for the highest-numbered guest boat across rounds.

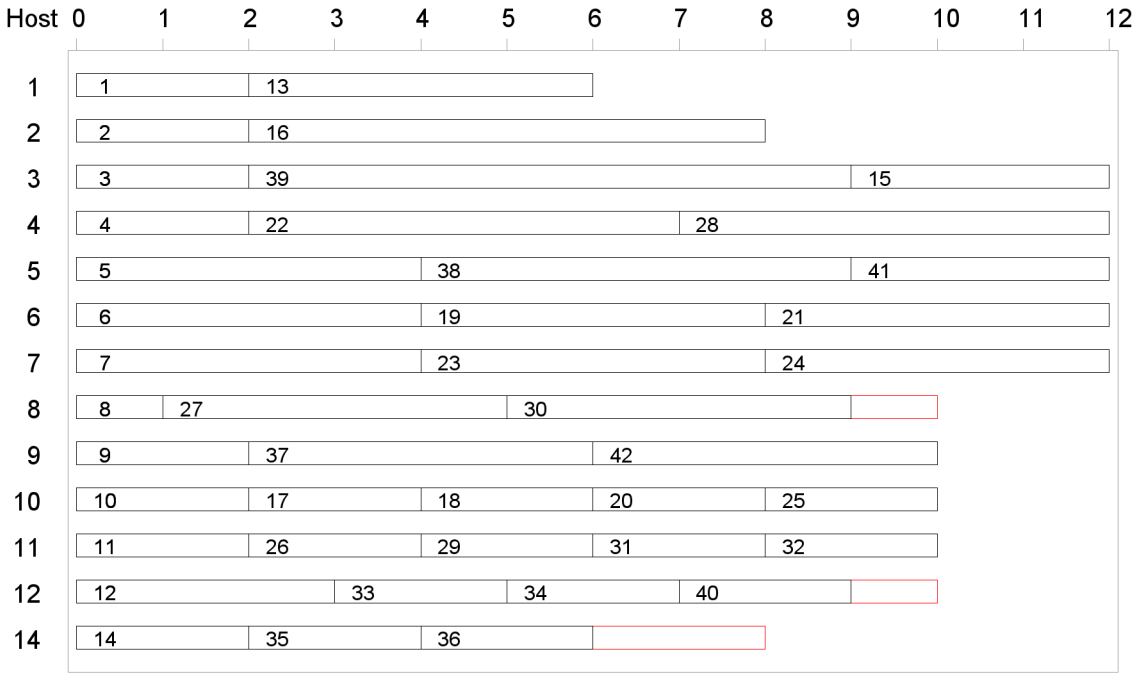
The following statements call PROC OPTMODEL to define the variables, specify the constraints, and solve the problem:

```
%let rounds=2;
%let numhosts=13;
proc optmodel;
   num numrounds = &rounds;
   set ROUNDS = 1..numrounds;
   num numhosts = &numhosts;
   set HOSTS = 1..numhosts;
   set BOATS;
   num numboats = card(BOATS);
   num capacity {BOATS};
   num crewsize {BOATS};
   num spareCapacity{hi in HOSTS} = capacity[hi] - crewsize[hi];
   /* Use the descending crew order for guests (seqno)
      rather than the actual boat id (boatnum)
      to help the performance of the PACK predicate. */
   read data capacities into BOATS=[seqno] capacity crewsize;

   /* Assume that the first numhosts boats are hosts,
      and process each round in turn.
      X is the host assigned to non-host i for round t. */
   var X {numhosts+1..numboats, ROUNDS} integer >= 1 <= numhosts;
   /* The load of the host boat. */
   var L {hi in HOSTS, ROUNDS} integer >= 0 <= spareCapacity[hi];

   /* Assign different hosts each round. */
   con AlldiffCon {i in numhosts+1..numboats}:
      alldiff({t in ROUNDS} X[i,t]);

   /* Two crews cannot meet more than once. */
   var M {i in numhosts+1..numboats-1, j in i+1..numboats, t in ROUNDS} binary;
   con ReifyCon {i in numhosts+1..numboats-1, j in i+1..numboats, t in ROUNDS}:
      reify(M[i,j,t], X[i,t] = X[j,t]);
   con Assign {i in numhosts+1..numboats-1, j in i+1..numboats}:
      sum {t in ROUNDS} M[i,j,t] <= 1;

   /* Honor capacities. */
   con PackCon {t in ROUNDS}:
      pack(
         {i in numhosts+1..numboats} X[i,t],
         {i in numhosts+1..numboats} crewsize[i],
         {h in HOSTS} L[h,t]
      );

   /* Break symmetries. */
   con SymmetryCon {t in 1..numrounds-1}:
      X[numboats,t] < X[numboats,t+1];

   solve with CLP / varselect=FIFO;
quit;
```

The two charts in Output 6.9.1 show the boat assignments for the first two rounds. The horizontal axis shows the load for each host boat. Slack capacity is highlighted in red.

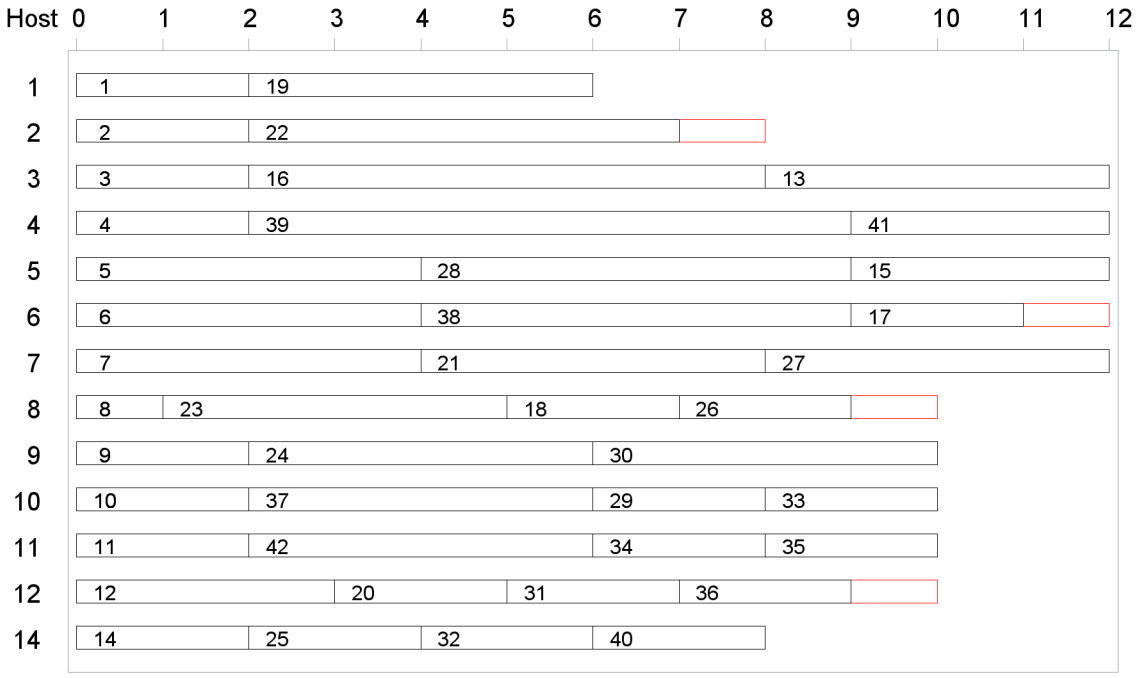
**Output 6.9.1** Gantt Chart: Boat Schedule by Round

**Schedule for Round 1**



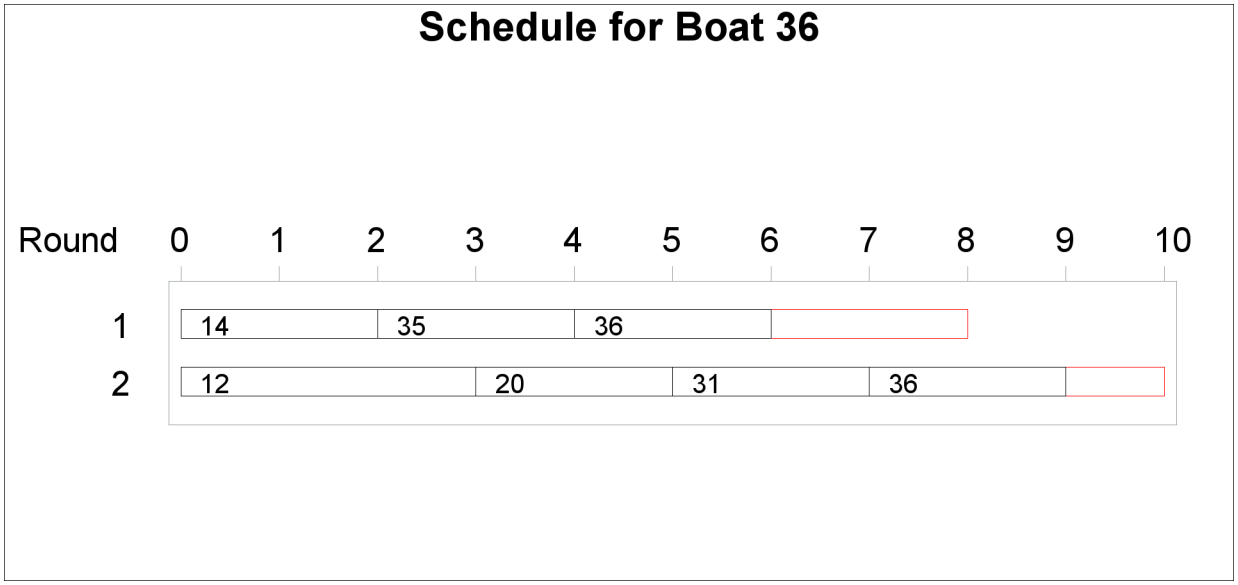
**Output 6.9.1** *continued*

**Schedule for Round 2**

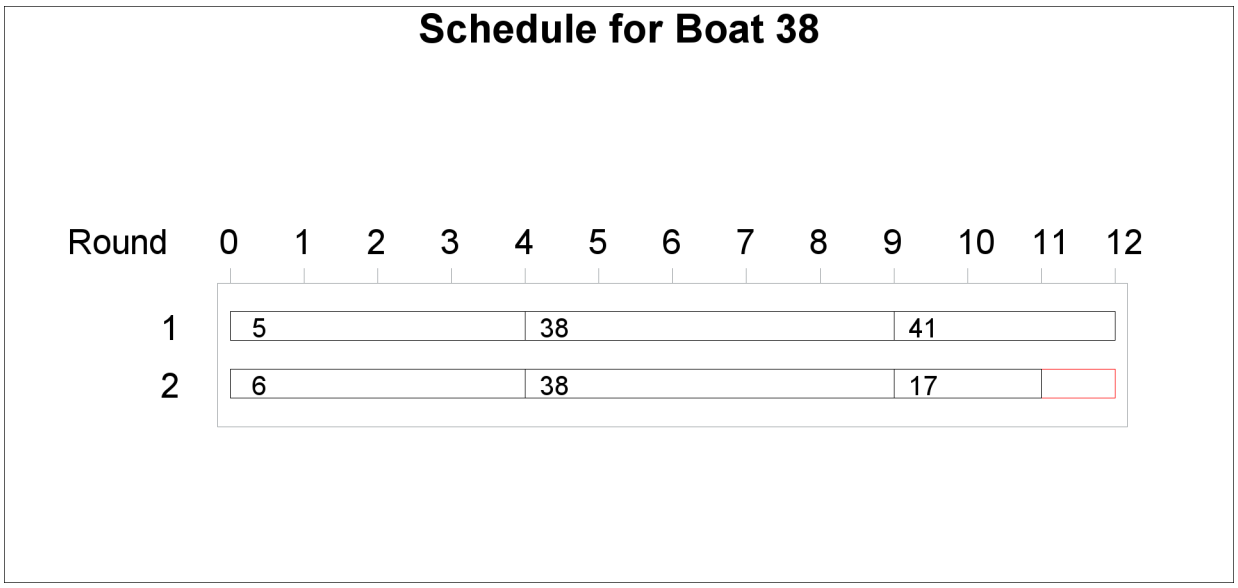


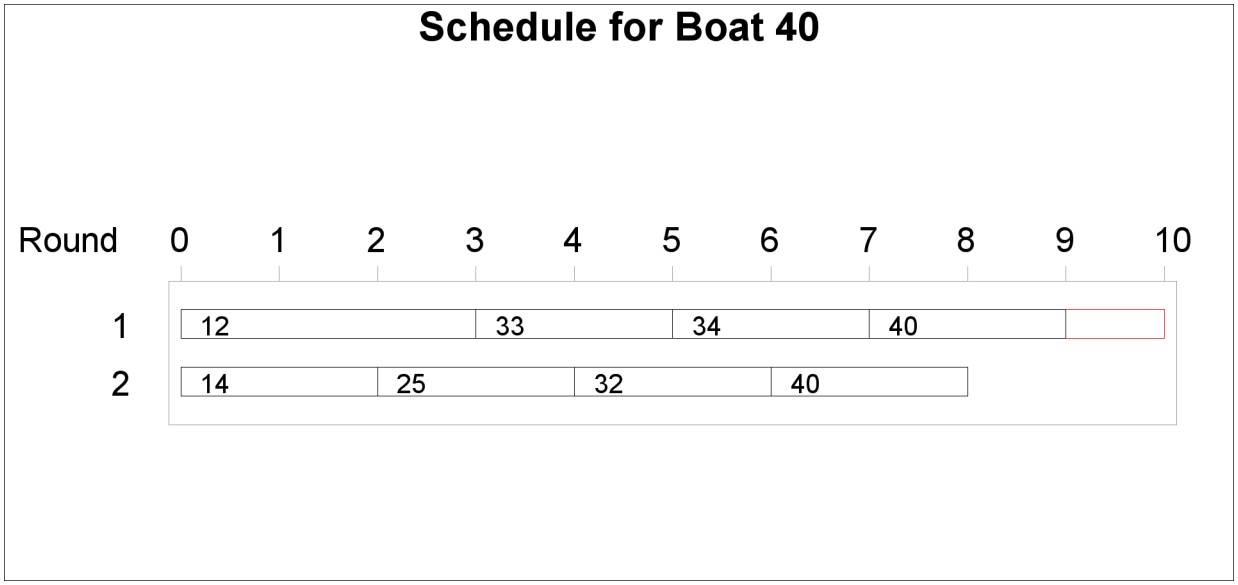

The charts in Output 6.9.2 break down the assignments by boat number for selected boats.

**Output 6.9.2** Gantt Chart: Host Boat Schedule by Round

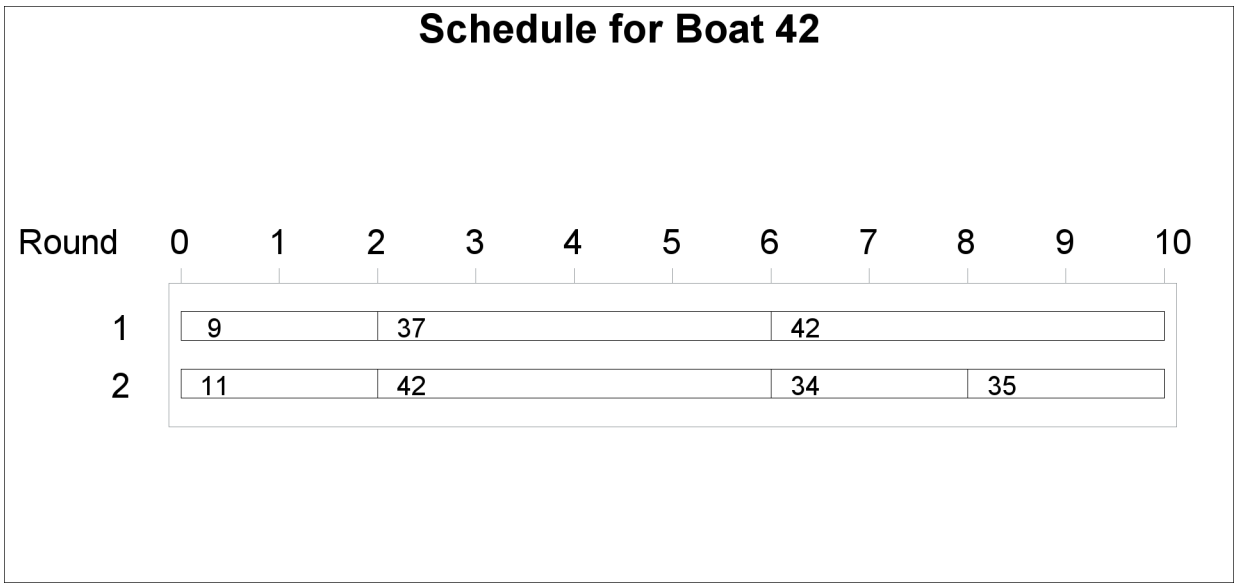


**Output 6.9.2** *continued*



**Output 6.9.2** *continued*




## References

Boussemart, F., Hemery, F., Lecoutre, C., and Sais, L. (2004). "Boosting Systematic Search by Weighting Constraints." In *ECAI 2004: Proceedings of the Sixteenth European Conference on Artificial Intelligence*, 146–150. Amsterdam: IOS Press.

Brualdi, R. A. (2010). *Introductory Combinatorics*. 5th ed. Englewood Cliffs, NJ: Prentice-Hall.

Dincbas, M., Simonis, H., and Van Hentenryck, P. (1988). "Solving the Car-Sequencing Problem in Constraint Logic Programming." In *Proceedings of the European Conference on Artificial Intelligence, ECAI-88*, edited by Y. Kodratoff, 290–295. London: Pitman.

Floyd, R. W. (1967). "Nondeterministic Algorithms." *Journal of the ACM* 14:636–644.

Frisch, A. M., Hnich, B., Kiziltan, Z., Miguel, I., and Walsh, T. (2002). "Global Constraints for Lexicographic Orderings." In *Proceedings of the Eighth International Conference on Principles and Practice of Constraint Programming (CP 2002)*, edited by P. Van Hentenryck, 93–108. London: Springer-Verlag.

Garey, M. R., and Johnson, D. S. (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York: W. H. Freeman.

Gravel, M., Gagne, C., and Price, W. L. (2005). "Review and Comparison of Three Methods for the Solution of the Car Sequencing Problem." *Journal of the Operational Research Society* 56:1287–1295.

Haralick, R. M., and Elliott, G. L. (1980). "Increasing Tree Search Efficiency for Constraint Satisfaction Problems." *Artificial Intelligence* 14:263–313.

Kumar, V. (1992). "Algorithms for Constraint-Satisfaction Problems: A Survey." *AI Magazine* 13:32–44.

Mackworth, A. K. (1977). "Consistency in Networks of Relations." *Artificial Intelligence* 8:99–118.

Meseguer, P., and Torras, C. (2001). "Exploiting Symmetries within Constraint Satisfaction Search." *Artificial Intelligence* 129:133–163.

Prestwich, S. D. (2001). "Balanced Incomplete Block Design as Satisfiability." In *Twelfth Irish Conference on Artificial Intelligence and Cognitive Science*, 189–198. Maynooth: National University of Ireland.

Riley, P., and Taalman, L. (2008). "Brainfreeze Puzzles." http://www.geekhaus.com/brainfreeze/piday2008.html.

Smith, B. M., Brailsford, S. C., Hubbard, P. M., and Williams, H. P. (1996). "The Progressive Party Problem: Integer Linear Programming and Constraint Programming Compared." *Constraints* 1:119–138.

Sokol, J. (2002). "Modeling Automobile Paint Blocking: A Time Window Traveling Salesman Problem." Ph.D. diss., Massachusetts Institute of Technology.

Solnon, C., Cung, V. D., Nguyen, A., and Artigues, C. (2008). "The Car Sequencing Problem: Overview of State-of-the-Art Methods and Industrial Case-Study of the ROADEF 2005 Challenge Problem." *European Journal of Operational Research* 191:912–927.

Trick, M. A. (2004). "Constraint Programming: A Tutorial." http://mat.gsia.cmu.edu/trick/cp.ppt.

Tsang, E. (1993). *Foundations of Constraint Satisfaction*. London: Academic Press.

Van Hentenryck, P. (1989). *Constraint Satisfaction in Logic Programming*. Cambridge, MA: MIT Press.

Van Hentenryck, P. (2002). "Constraint and Integer Programming in OPL." *INFORMS Journal on Computing* 14:345–372.

Waltz, D. L. (1975). "Understanding Line Drawings of Scenes with Shadows." In *The Psychology of Computer Vision*, edited by P. H. Winston, 19–91. New York: McGraw-Hill.

# Subject Index

# Syntax Index