SAS/OR® 15.1 User's Guide
Constraint Programming
The CLP Procedure

This document is an individual chapter from *SAS/OR® 15.1 User's Guide: Constraint Programming*.

The correct bibliographic citation for this manual is as follows: SAS Institute Inc. 2018. *SAS/OR® 15.1 User's Guide: Constraint Programming*. Cary, NC: SAS Institute Inc.

SAS/OR® 15.1 User's Guide: Constraint Programming

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

November 2018

Chapter 3
The CLP Procedure

Contents

Overview: CLP Procedure

The CLP procedure is a finite-domain constraint programming solver for constraint satisfaction problems (CSPs) with linear, logical, global, and scheduling constraints. In addition to having an expressive syntax for representing CSPs, the CLP procedure features powerful built-in consistency routines and constraint propagation algorithms, a choice of nondeterministic search strategies, and controls for guiding the search mechanism that enable you to solve a diverse array of combinatorial problems.

You can also solve standard constraint satisfaction problems via the constraint programming solver in PROC OPTMODEL. For more information, see the constraint programming solver chapter in *SAS/OR User's Guide: Mathematical Programming*.

The Constraint Satisfaction Problem

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.

More formally, a CSP can be defined as a triple $\langle X, D, C \rangle$:

- $X = \{x_1, \ldots, x_n\}$ 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 need not represent consecutive integers. For example, the domain of a variable could be the set of all *even* numbers in the interval [0, 100]. A domain does not even need to be totally numeric. In fact, in a scheduling problem with resources, the values are typically multidimensional. For example, an activity can be considered as a variable, and each element of the domain would be an *n-tuple* that represents a start time for the activity and one or more resources that must be assigned to the activity that corresponds to the start time.

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.

The CLP procedure can be used to find one or more (and in some instances, all) solutions to a CSP with linear, logical, global, and scheduling constraints. The numeric components of all variable domains are assumed to be integers.

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, it is oriented toward *recovering* from failures rather than *avoiding* them to begin with. The search space is reduced only after detection of a failure, and the performance of this technique is drastically reduced with increasing problem size. 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 in solving 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.

An effective consistency technique for handling resource capacity constraints is edge finding (Applegate and Cook 1991). Edge-finding techniques reason about the processing order of a set of activities that require a given resource or set of resources. Some of the earliest work related to edge finding can be attributed to Carlier and Pinson (1989), who successfully solved MT10, a well-known 10×10 job shop problem that had remain unsolved for over 20 years (Muth and Thompson 1963).

Constraint propagation is characterized by the extent of propagation (also referred to as 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 over domain propagation 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, implemented as follows. Whenever a node is visited, constraint propagation is carried out 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, whereas performing consistency on variables already selected reduces this 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 strategies, 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 more 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 technique as providing substantial improvement for a significant class of problems. Another popular technique 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 to detect 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).

The CLP Procedure

The CLP procedure is a finite-domain constraint programming solver for CSPs. In the context of the CLP procedure, CSPs can be classified into the following two types, which are determined by specification of the relevant output data set:

- A *standard CSP* is characterized by integer variables, linear constraints, array-type constraints, global constraints, and reified constraints. In other words, X is a finite set of integer variables, and C can contain linear, array, global, or logical constraints. Specifying the OUT= option in the PROC CLP statement indicates to the CLP procedure that the CSP is a standard-type CSP. As such, the procedure expects only VARIABLE, ALLDIFF, CUMULATIVE, ELEMENT, GCC, LEXICO, LINCON, PACK, REIFY, ARRAY, and FOREACH statements. You can also specify a Constraint data set by using the CONDATA= option in the PROC CLP statement instead of, or in combination with, VARIABLE and LINCON statements.

- A *scheduling CSP* is characterized by activities, temporal constraints, and resource requirement constraints. In other words, X is a finite set of activities, and C is a set of temporal constraints and resource requirement constraints. Specifying one of the SCHEDULE=, SCHEDRES=, or SCHEDTIME= options in the PROC CLP statement indicates to the CLP procedure that the CSP is a scheduling-type CSP. As such, the procedure expects only ACTIVITY, RESOURCE, REQUIRES, and SCHEDULE statements. You can also specify an Activity data set by using the ACTDATA= option in the PROC CLP statement instead of, or in combination with, the ACTIVITY, RESOURCE, and REQUIRES statements. You can define activities by using the Activity data set or the ACTIVITY statement. Precedence relationships between activities must be defined using the ACTDATA= data set. You can define resource requirements of activities by using the Activity data set or the RESOURCE and REQUIRES statements.

The output data sets contain any solutions determined by the CLP procedure. For more information about the format and layout of the output data sets, see the sections "Solution Data Set" on page 46 and "Schedule Data Set" on page 50.

Consistency Techniques

The CLP procedure 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. For the scheduling CSPs, the CLP procedure uses a forward-checking algorithm, an arc-consistency routine for maintaining consistency between unassigned activities, and energetic-based reasoning methods for resource-constrained scheduling that feature the edge-finder algorithm (Applegate and Cook 1991). You can elect to turn off some of these consistency techniques in the interest of performance.

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 PROC CLP statement defines the variable selection strategy for a standard CSP. The default strategy is the dynamic MINR strategy, which selects the variable with the smallest range. The ACTSELECT= option in the SCHEDULE statement defines the activity selection strategy for a scheduling CSP. The default strategy is the RAND strategy, which selects an activity at random from the set of activities that begin prior to the earliest early finish time. This strategy was proposed by Nuijten (1994).

Assignment Strategy

After a variable or an activity has been selected, the assignment strategy dictates the value that is assigned to it. For variables, the assignment strategy is specified with the VARASSIGN= option in the PROC CLP statement. The default assignment strategy selects the minimum value from the domain of the selected variable. For activities, the assignment strategy is specified with the ACTASSIGN= option in the SCHEDULE statement. The default strategy of RAND assigns the time to the earliest start time, and the resources are chosen randomly from the set of resource assignments that support the selected start time.

Getting Started: CLP Procedure

The following examples illustrate the use of the CLP procedure in the formulation and solution of two well-known logical puzzles in the constraint programming community.

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:

$$\begin{array}{r} S\,E\,N\,D \\ +\,M\,O\,R\,E \\ \hline M\,O\,N\,E\,Y \end{array}$$

You can use the CLP procedure to formulate this problem as a CSP by representing each of the letters in the expression with an integer variable. The domain of each variable is the set of digits 0 through 9. The VARIABLE statement identifies the variables in the problem. The DOM= option defines the default domain for all the variables to be [0,9]. The OUT= option identifies the CSP as a standard type. The LINCON statement defines the linear constraint SEND + MORE = MONEY, and the restrictions that S and M cannot take the value zero. (Alternatively, you can simply specify the domain for S and M as [1,9] in the VARIABLE statement.) Finally, the ALLDIFF statement is specified to enforce the condition that the assignment of digits should be unique. The complete representation, using the CLP procedure, is as follows:

```
proc clp dom=[0,9]                    /* Define the default domain */
         out=out;                     /* Name the solution datset  */
   var S E N D M O R E M O N E Y;     /* Declare the variables     */
   lincon                             /* Linear constraints        */
                                      /* SEND + MORE = MONEY       */
      1000*S + 100*E + 10*N + D + 1000*M + 100*O + 10*R + E
      =
      10000*M + 1000*O + 100*N  + 10*E + Y,
      S<>0,                           /* No leading zeros          */
      M<>0;
   alldiff();      /* All variables have pairwise distinct values*/
run;
```

The Solution data set that is produced by the CLP procedure is shown in Output 3.1.

Output 3.1 Solution to SEND + MORE = MONEY

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

The unique solution to the problem that the CLP procedure finds is as follows:

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

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 attack each other. The CLP procedure 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 A of dimension N, where $A[i]$ is the row number of the queen in column i. Since 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 should be true for all i and j:

$$A[j] - A[i] <> j - i$$

and

$$A[j] - A[i] <> i - j$$

In other words,

$$A[i] - i <> A[j] - j$$

and

$$A[i] + i <> A[j] + j$$

Hence, the $(A[i] + i)$'s are pairwise distinct, and the $(A[i] - i)$'s are pairwise distinct.

These two conditions, in addition to the one requiring that the $A[i]$'s be pairwise distinct, can be formulated using the FOREACH statement.

One possible such CLP formulation is presented as follows:

```
proc clp out=out
         varselect=fifo; /* Variable Selection Strategy                 */
   array A[8] (A1-A8);   /* Define the array A                          */
   var (A1-A8)=[1,8];    /* Define each of the variables in the array */
                         /* Initialize domains                          */
   /* A[i] is the row number of the queen in column i*/
   foreach(A, DIFF,  0); /* A[i] 's are pairwise distinct */
   foreach(A, DIFF, -1); /* A[i] - i 's are pairwise distinct */
   foreach(A, DIFF,  1); /* A[i] + i 's are pairwise distinct */
run;
```

The ARRAY statement is required when you are using a FOREACH statement, and it defines the array A in terms of the eight variables A1–A8. The domain of each of these variables is explicitly specified in the VARIABLE statement to be the digits 1 through 8 since they represent the row number on an 8×8 board. FOREACH(A, DIFF, 0) represents the constraint that the $A[i]$'s are different. FOREACH(A, DIFF, –1) represents the constraint that the $(A[i] - i)$'s are different, and FOREACH(A, DIFF, 1) represents the constraint that the $(A[i] + i)$'s are different. The VARSELECT= option specifies the variable selection strategy to be first-in-first-out, the order in which the variables are encountered by the CLP procedure.

The following statements display the Solution data set shown in Output 3.2:

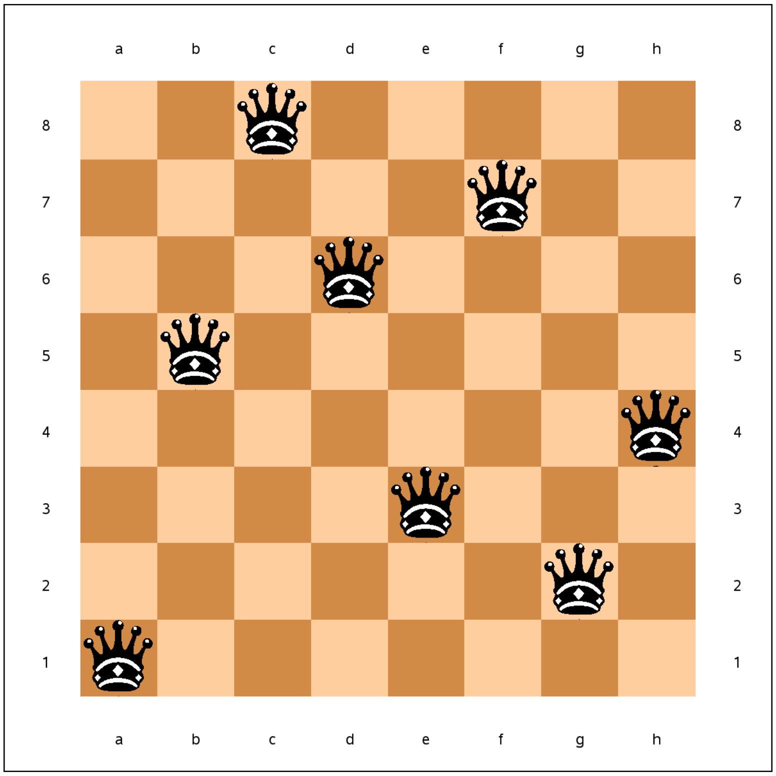
```
proc print data=out noobs label;
   label A1=a A2=b A3=c A4=d
         A5=e A6=f A7=g A8=h;
run;
```

Output 3.2 A Solution to the Eight Queens Problem

a	b	c	d	e	f	g	h
1	5	8	6	3	7	2	4

The corresponding solution to the Eight Queens problem is displayed in Figure 3.1.

Figure 3.1 A Solution to the Eight Queens Problem



Syntax: CLP Procedure

The following statements are used in PROC CLP:

PROC CLP *options* ;
  ACTIVITY *activity specifications* ;
  ALLDIFF *alldiff constraints* ;
  CUMULATIVE *cumulative constraints* ;
  ARRAY *array specifications* ;
  ELEMENT *element constraints* ;
  FOREACH *foreach constraints* ;
  GCC *global cardinality constraints* ;
  LEXICO *lexicographic ordering constraints* ;
  LINCON *linear constraints* ;
  OBJ *objective function options* ;
  PACK *bin packing constraints* ;
  REIFY *reify constraints* ;
  REQUIRES *resource requirement constraints* ;
  RESOURCE *resource specifications* ;
  SCHEDULE *schedule options* ;
  VARIABLE *variable specifications* ;

Functional Summary

The statements and options available in PROC CLP are summarized by purpose in Table 3.1.

Table 3.1 Functional Summary

Description	Statement	Option
Assignment Strategy Options		
Specifies the variable assignment strategy	PROC CLP	VARASSIGN=
Specifies the activity assignment strategy	SCHEDULE	ACTASSIGN=
Data Set Options		
Specifies the activity input data set	PROC CLP	ACTDATA=
Specifies the constraint input data set	PROC CLP	CONDATA=
Specifies the solution output data set	PROC CLP	OUT=
Specifies the resource input data set	PROC CLP	RESDATA=
Specifies the resource assignment data set	PROC CLP	SCHEDRES=
Specifies the time assignment data set	PROC CLP	SCHEDTIME=
Specifies the schedule output data set	PROC CLP	SCHEDULE=
Domain Options		
Specifies the global domain of all variables	PROC CLP	DOMAIN=
Specifies the domain for selected variables	VARIABLE	
General Options		
Specifies the upper bound on time (seconds)	PROC CLP	MAXTIME=
Suppresses preprocessing	PROC CLP	NOPREPROCESS
Permits preprocessing	PROC CLP	PREPROCESS
Specifies the units of MAXTIME	PROC CLP	TIMETYPE=
Implicitly defines constraint data set variables	PROC CLP	USECONDATAVARS=
Objective Function Options		
Specifies the lower bound for the objective	OBJ	LB=
Specifies the tolerance for the search	OBJ	TOL=
Specifies the upper bound for the objective	OBJ	UB=
Output Control Options		
Finds all possible solutions	PROC CLP	FINDALLSOLNS
Specifies the number of solution attempts	PROC CLP	MAXSOLNS=
Indicates progress in log	PROC CLP	SHOWPROGRESS
Scheduling CSP-Related Statements		
Defines activity specifications	ACTIVITY	
Defines resource requirement specifications	REQUIRES	
Defines resource specifications	RESOURCE	
Defines scheduling parameters	SCHEDULE	

Table 3.1 *continued*

Description	Statement	Option
Scheduling: Resource Constraints		
Specifies the edge-finder consistency routines	SCHEDULE	EDGEFINDER=
Specifies the not-first edge-finder extension	SCHEDULE	NOTFIRST=
Specifies the not-last edge-finder extension	SCHEDULE	NOTLAST=
Scheduling: Temporal Constraints		
Specifies the activity duration	ACTIVITY	(DUR=)
Specifies the activity finish lower bound	ACTIVITY	(FGE=)
Specifies the activity finish upper bound	ACTIVITY	(FLE=)
Specifies the activity start lower bound	ACTIVITY	(SGE=)
Specifies the activity start upper bound	ACTIVITY	(SLE=)
Specifies the schedule duration	SCHEDULE	DURATION=
Specifies the schedule finish	SCHEDULE	FINISH=
Specifies the schedule start	SCHEDULE	START=
Scheduling: Search Control Options		
Specifies the number of allowable dead ends per restart	PROC CLP	DPR=
Specifies the number of search restarts	PROC CLP	RESTARTS=
Selection Strategy Options		
Specifies the variable selection strategy	PROC CLP	VARSELECT=
Specifies the activity selection strategy	SCHEDULE	ACTSELECT=
Specifies variable selection strategies for evaluation	PROC CLP	EVALVARSEL=
Specifies activity selection strategies for evaluation	SCHEDULE	EVALACTSEL=
Enables time limit updating for strategy evaluation	PROC CLP	DECRMAXTIME
Standard CSP Statements		
Specifies the alldifferent constraints	ALLDIFF	
Specifies the array specifications	ARRAY	
Specifies the element constraints	ELEMENT	
Specifies the foreach constraints	FOREACH	
Specifies the global cardinality constraints	GCC	
Specifies the lexicographic ordering constraints	LEXICO	
Specifies the linear constraints	LINCON	
Specifies the bin packing constraints	PACK	
Specifies the reified constraints	REIFY	
Defines the variable specifications	VARIABLE	

PROC CLP Statement

PROC CLP *options* **;**

The PROC CLP statement invokes the CLP procedure. You can specify options to control a variety of search parameters that include selection strategies, assignment strategies, backtracking strategies, maximum running time, and number of solution attempts. You can specify the following options:

ACTDATA=*SAS-data-set*

ACTIVITY=*SAS-data-set*

identifies the input SAS data set that defines the activities and temporal constraints. The temporal constraints consist of time-alignment-type constraints and precedence-type constraints. The format of the ACTDATA= data set is similar to that of the Activity data set used by the CPM procedure in SAS/OR software. You can also specify the activities and time alignment constraints directly by using the ACTIVITY statement without the need for a data set. The CLP procedure enables you to define activities by using a combination of the two specifications.

CONDATA=*SAS-data-set*

identifies the input data set that defines the constraints, variable types, and variable bounds and an objective function. The CONDATA= data set provides support for linear constraints only.

The format of the CONDATA= data set is similar to that of the DATA= data set used by the LP procedure in SAS/OR software.

You can also specify the linear constraints in-line by using the LINCON statement. The CLP procedure enables you to define constraints by using a combination of the two specifications. When defining constraints, you must define the variables by using a VARIABLE statement or implicitly define them by specifying the USECONDATAVARS= option when using the CONDATA= data set. You can define variable bounds by using the VARIABLE statement, and any such definitions override the implicit definitions of variables in the CONDATA= data set.

DECRMAXTIME

dynamically decreases the maximum solution time in effect during evaluation of the selection strategy. The DECRMAXTIME option is effective only when the EVALACTSEL= option or the EVALVARSEL= option is specified. Initially, the maximum solution time is the value specified by the MAXTIME= option. Whenever a solution is found with the current activity or variable selection strategy, the value of MAXTIME for future attempts is reduced to the current solution time. The DECRMAXTIME option thus provides a sense of which strategy is the fastest for a given problem. However, you must use caution when comparing the strategies, because the individual strategy results might pertain to different time limits.

By default, the DECRMAXTIME option is disabled; each activity or variable selection strategy is given the amount of time specified by the MAXTIME= option.

DM=*m*

specifies the dead-end multiplier for the scheduling CSP. The dead-end multiplier is used to determine the number of dead ends that are permitted before triggering a complete restart of the search technique in a scheduling environment. The number of dead ends is the product of the dead-end multiplier, *m*, and the number of unassigned activities. The default value is 0.15. This option is valid only with the SCHEDULE= option.

DOMAIN=*[lb, ub]*

DOM=*[lb, ub]*

specifies the global domain of all variables to be the closed interval [*lb*, *ub*]. You can override the global domain for a variable with a VARIABLE statement or the CONDATA= data set. The default domain is [0,∞].

DPR=*n*

specifies an upper bound on the number of dead ends that are permitted before PROC CLP restarts or terminates the search, depending on whether or not a randomized search strategy is used. In the case of a nonrandomized strategy, *n* is an upper bound on the number of allowable dead ends before terminating. In the case of a randomized strategy, *n* is an upper bound on the number of allowable dead ends before restarting the search. The DPR= option has priority over the DM= option.

EVALVARSEL*<=(keyword(s))>*

evaluates specified variable selection strategies by attempting to find a solution with each strategy. You can specify any combination of valid variable selection strategies in a space-delimited list enclosed in parentheses. If you do not specify a list, all available strategies are evaluated in alphabetical order, except that the default strategy is evaluated first. Descriptions of the available selection strategies are provided in the discussion of the VARSELECT= option.

When the EVALVARSEL= option is in effect, the MAXTIME= option must also be specified. By default, the value specified for the MAXTIME= option is used as the maximum solution time for each variable selection strategy. When the DECRMAXTIME option is specified and a solution has been found, the value of the MAXTIME= option is set to the solution time of the current solution.

After the CLP procedure has attempted to find a solution with a particular strategy, it proceeds to the next strategy in the list. For this reason, the VARSELECT=, ALLSOLNS, and MAXSOLNS= options are ignored when the EVALVARSEL= option is in effect. All solutions that are found during the evaluation process are saved in the output data set that you specify in the OUT= option.

The macro variable _ORCLPEVS_ provides more information that is related to the evaluation of each variable selection strategy. The fastest variable selection strategy is indicated in the macro variable _ORCLP_, provided that either at least one solution is found or the problem is found to be infeasible. See "Macro Variable _ORCLP_" on page 53 for more information about the _ORCLP_ macro variable; see "Macro Variable _ORCLPEVS_" on page 55 for more information about the _ORCLPEVS_ macro variable.

FINDALLSOLNS

ALLSOLNS

FINDALL

attempts to find all possible solutions to the CSP. When a randomized search strategy is used, it is possible to rediscover the same solution and end up with multiple instances of the same solution. This is currently the case when you are solving a scheduling CSP. Therefore, this option is ignored when you are solving a scheduling CSP.

MAXSOLNS=*n*

specifies the number of solution attempts to be generated for the CSP. The default value is 1. It is important to note, especially in the context of randomized strategies, that an attempt could result in no solution, given the current controls on the search mechanism, such as the number of restarts and the number of dead ends permitted. As a result, the total number of solutions found might not match the MAXSOLNS= parameter.

MAXTIME=*n*

specifies an upper bound on the number of seconds that are allocated for solving the problem. The type of time, either CPU time or real time, is determined by the value of the TIMETYPE= option. The default type is CPU time. The time specified by the MAXTIME= option is checked only once at the end of each iteration. Therefore, the actual running time can be longer than that specified by the MAXTIME= option. The difference depends on how long the last iteration takes. The default value of MAXTIME= is ∞. If you do not specify this option, the procedure does not stop based on the amount of time elapsed.

NOPREPROCESS

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

OUT=*SAS-data-set*

identifies the output data set that contains one or more solutions to a standard CSP, if one exists. Each observation in the OUT= data set corresponds to a solution of the CSP. You can control the number of solutions that are generated by specifying the MAXSOLNS= option in the PROC CLP statement.

PREPROCESS

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

RESDATA=*SAS-data-set*

RESIN=*SAS-data-set*

identifies the input data set that defines the resources and their attributes such as capacity and resource pool membership. This information can be used in lieu of, or in combination with, the RESOURCE statement.

RESTARTS=*n*

specifies the number of restarts of the randomized search technique before terminating the procedure. The default value is 3.

SCHEDRES=*SAS-data-set*

identifies the output data set that contains the solutions to scheduling CSPs. This data set contains the resource assignments of activities.

SCHEDTIME=*SAS-data-set*

identifies the output data set that contains the solutions to scheduling CSPs. This data set contains the time assignments of activities.

SCHEDULE=*SAS-data-set*

SCHEDOUT=*SAS-data-set*

identifies the output data set that contains the solutions to a scheduling CSP, if any exist. This data set contains both the time and resource assignment information. There are two types of observations identified by the value of the OBSTYPE variable: observation with OBSTYPE= “TIME” corresponds to time assignment, and observation with OBSTYPE= “RESOURCE” corresponds to resource assignment. The maximum number of solutions can be controlled by using the MAXSOLNS= option in the PROC CLP statement.

SHOWPROGRESS

prints a message to the log whenever a solution has been found. When a randomized strategy is used, the number of restarts and dead ends that were required are also printed to the log.

TIMETYPE=CPU | REAL

specifies whether CPU time or real time is used for evaluation of the MAXTIME= option and the reporting of timing values in the macro variables. The default value of this option is CPU.

USECONDATAVARS=0 | 1

specifies whether the numeric variables in the CONDATA= data set, with the exception of any reserved variables, are implicitly defined or not. A value of 1 indicates they are implicitly defined, in which case a VARIABLE statement is not necessary to define the variables specified in the data set. The default value is 0. Currently, _RHS_ is the only reserved numeric variable.

VARASSIGN=MAX | MIN

specifies the value selection strategy. Currently, there are two 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

The default value selection strategy is MIN. To assign activities, use the ACTASSIGN= option in the SCHEDULE statement.

VARSELECT=*keyword*

specifies the variable selection strategy. The strategy can 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 are strategies which exploit information from previous states of the search process as well as the current state (Boussemart et al. 2004)

Static strategies are as follows:

- FIFO, which uses the first-in-first-out ordering of the variables as encountered by the procedure
- MAXCS, which selects the variable with the maximum number of constraints (degree)

Dynamic strategies are as follows:

- DOMDDEG, which selects the variable with the smallest ratio of domain size by dynamic degree
- DOMDEG, which selects the variable with the smallest ratio of domain size by degree
- MAXC, which selects the variable with the largest number of active constraints (dynamic degree)
- MINR, which selects the variable with the smallest range (that is, the minimum value of upper bound minus lower bound)
- MINRMAXC, which selects the variable with the smallest range, breaking ties by selecting one with the largest number of active constraints

Conflict-directed strategies are as follows:

- DOMWDEG, which selects the variable with the smallest ratio of domain size by weighted degree
- WDEG, which selects the variable with the largest weighted degree

The dynamic strategies embody the "fail first principle" (FFP) of Haralick and Elliott (1980), which suggests that "To succeed, try first where you are most likely to fail." The degree of a variable is the number of constraints in which the variable appears. The dynamic degree of a variable is the degree of the variable with respect to the current set of active constraints. Boussemart et al. (2004) introduced the concept of weighted degree, which takes into account dead ends during searches. Weights are associated with constraints and variables. The weight of a constraint is initially set to 1 and incremented each time the constraint is responsible for a dead end. The weight of a variable is equal to the sum of the weights of the active constraints that it is associated with. The default variable selection strategy is MINR. To set the strategy for selecting activities, use the ACTSELECT= option in the SCHEDULE statement.

ACTIVITY Statement

ACTIVITY *specification-1* < . . . *specification-n* > **;**

An ACTIVITY *specification* can be one of the following types:

activity < = (< **DUR=** > *duration* < *altype=aldate* . . . >) >

(activity_list) < = (< **DUR=** > *duration* < *altype=aldate* . . . >) >

where *duration* is the activity duration and *altype* is a keyword that specifies an alignment-type constraint on the activity (or activities) with respect to the value given by *aldate*.

The ACTIVITY statement defines one or more activities and the attributes of each activity, such as the duration and any temporal constraints of the time-alignment-type. The activity duration can take nonnegative integer values. The default duration is 0.

Valid *altype* keywords are as follows:

- **SGE**, start greater than or equal to *aldate*
- **SLE**, start less than or equal to *aldate*
- **FGE**, finish greater than or equal to *aldate*
- **FLE**, finish less than or equal to *aldate*

You can specify any combination of the preceding keywords. For example, to define activities A1, A2, A3, B1, and B3 with duration 3, and to set the start time of these activities equal to 10, specify the following:

```
activity (A1-A3 B1 B3) = ( dur=3 sge=10 sle=10 );
```

If an activity appears in more than one ACTIVITY statement, only the first activity definition is honored. Additional specifications are ignored.

You can alternatively use the ACTDATA= data set to define activities, durations, and temporal constraints. In fact, you can specify both an ACTIVITY statement and an ACTDATA= data set. You must use an ACTDATA= data set to define precedence-related temporal constraints. One of SCHEDULE=, SCHEDRES=, or SCHEDTIME= must be specified when the ACTIVITY statement is used.

ALLDIFF Statement

ALLDIFF *(variable_list-1)* < . . . *(variable_list-n)* > **;**

ALLDIFFERENT *(variable_list-1)* < . . . *(variable_list-n)* > **;**

The ALLDIFF statement can have multiple specifications. Each specification defines a unique global constraint on a set of variables, requiring all of them to be different from each other. A global constraint is equivalent to a conjunction of elementary constraints.

For example, the statements

```
var (X1-X3) A B;
alldiff (X1-X3) (A B);
```

are equivalent to

$$\begin{aligned} X1 &\neq X2 \text{ AND} \\ X2 &\neq X3 \text{ AND} \\ X1 &\neq X3 \text{ AND} \\ A &\neq B \end{aligned}$$

If the variable list is empty, the ALLDIFF constraint applies to all the variables.

ARRAY Statement

ARRAY *specification-1* < . . . *specification-n* > **;**

An ARRAY *specification* is in a form as follows:

name[dimension](variables)

The ARRAY statement is used to associate a *name* with a list of *variables*. Each of the *variables* in the variable list must be defined using a VARIABLE statement or implicitly defined using the CONDATA= data set. The ARRAY statement is required when you are specifying a constraint by using the FOREACH statement.

CUMULATIVE Statement

CUMULATIVE *cumulative_constraint-1* < . . . *cumulative_constraint-n* > **;**

Each *cumulative_constraint* is specified in the following form, where the options can be specified in any order and are subject to the restrictions that follow:

(<**START=(***list***)**> <**DURATION=(***list***)**> <**END=(***list***)**> <**DEMAND=(***list***)**> <**CAPACITY=***capacity*> **)**

- You must specify at least two of the START=, DURATION=, and END= options.
- At least one of the START=, DURATION=, and END= options must specify a list of numeric variables.

- The number of values in the *lists* that are specified in the START=, DURATION=, END=, and DEMAND= options must be the same.

The CUMULATIVE statement specifies one or more cumulative (scheduling) constraints. A cumulative constraint conveys that a collection of tasks (activities) is to be executed on a resource that has limited capacity. Each task is defined by its start time, duration (processing time), end time (finish time), and demand (resource usage). A task i is said to overlap a time point t if and only if its start time is less than or equal to t and its end time is strictly greater than t. The cumulative constraint enforces that at each point in time, the accumulated demand (resource usage) of the tasks that overlap the time point does not exceed the specified capacity of the resource.

A cumulative constraint also enforces the following equation for all tasks i:

$$\text{start\_time}[i] + \text{duration}[i] = \text{end\_time}[i]$$

You can specify the following options (you must specify at least two of the START=, END=, and DURATION= options):

START=(*variables* **|** *integers* **)**
specifies a list of task starting times. If you specify this option, the number of *variables* or *integers* must match the number of *variables* or *integers* that are specified in each of the specified END=, DURATION=, and DEMAND= options.

DURATION=(*variables* **|** *nonnegative-integers* **)**
DUR=(*variables* **|** *nonnegative-integers* **)**
specifies a list of task processing durations. If you specify this option, the number of *variables* or *nonnegative-integers* must match the number of *variables* or *integers* that are specified in each of the specified START=, END=, and DEMAND= options.

END=(*variables* **|** *integers* **)**
specifies a list of task ending times. If you specify this option, the number of *variables* or *integers* must match the number of *variables* or *integers* that are specified in each of the specified START=, DURATION=, and DEMAND= options.

DEMAND=(*variables* **|** *nonnegative-integers* **)**
HEIGHT=(*variables* **|** *nonnegative-integers* **)**
specifies a list of task demands. If you specify this option, the number of *variables* or *nonnegative-integers* must match the number of *variables* or *integers* that are specified in each of the specified START=, END=, and DURATION= options. If you omit this option, the demand of each corresponding task is 1.

CAPACITY=*variable* **|** *nonnegative-integer*
LIMIT=*variable* **|** *nonnegative-integer*
specifies the capacity limit of the corresponding resource. If you omit this option, the capacity of each corresponding resource is 1.

ELEMENT Statement

ELEMENT *element_constraint-1 <. . . element_constraint-n>* **;**

An *element_constraint* is specified in the following form:

(index variable, (integer list), variable)

The ELEMENT statement specifies one or more element constraints. An element constraint enables you to define dependencies, not necessarily functional, between variables. The statement

ELEMENT$(I, (L), V)$

sets the variable V to be equal to the Ith element in the list L. The list of integers $L = (v_1, ..., v_n)$ is a list of values that are potentially assigned to the variable V and are not necessarily distinct. 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.

An element constraint enforces the following propagation rules:

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

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

The following statements use the element constraint to implement the quadratic function $y = x^2$:

```
proc clp out=clpout;
   var x=[1,5] y=[1,25];
   element (x,(1, 4, 9, 16, 25), y);
run;
```

An element constraint is equivalent to a conjunction of reified and linear constraints. For example, the preceding statements are equivalent to:

```
proc clp out=clpout;
        var x=[1,5] y=[1,25] (R1-R5)=[0,1];
        reify R1: (x=1);
        reify R1: (y=1);
        reify R2: (x=2);
        reify R2: (y=4);
        reify R3: (x=3);
        reify R3: (y=9);
        reify R4: (x=4);
        reify R4: (y=16);
        reify R5: (x=5);
        reify R5: (y=25);
        lincon R1 + R2 + R3 + R4 + R5 = 1;
run;
```

Element constraints can also be used 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 model this by using two element constraints and an artificial index variable:

```
element (i, ( -5, -3, -1, 1, 3,  5), x)
        (i, ( 25,  9,  1, 1, 9, 25), y);
```

The list of values L can also be specified by using a convenient syntax of the form *start* TO *end* or *start* TO *end* BY *increment*. For example, the previous element specification is equivalent to:

```
element (i, ( -5 to 5 by 2), x)
        (i, ( 25,  9,  1, 1, 9, 25), y);
```

FOREACH Statement

FOREACH *(array, type, < offset>)* **;**

where *array* must be defined by using an ARRAY statement, *type* is a keyword that determines the type of the constraint, and *offset* is an integer.

The FOREACH statement iteratively applies a constraint over an array of variables. The type of the constraint is determined by *type*. Currently, the only valid *type* keyword is DIFF. The optional *offset* parameter is an integer and is interpreted in the context of the constraint type. The default value of *offset* is zero.

The FOREACH statement that corresponds to the DIFF keyword iteratively applies the following constraint to each pair of variables in the array:

$$variable\_i + offset \times i \neq variable\_j + offset \times j \quad \forall\, i \neq j,\ i, j = 1, \ldots, array\_dimension$$

For example, the constraint that all $(A[i] - i)$'s are pairwise distinct for an array A is expressed as

```
foreach (A, diff, -1);
```

GCC Statement

GCC *global_cardinality_constraint-1* < *. . . global_cardinality_constraint-n* > **;**

where *global_cardinality_constraint* is specified in the following form:

(*variables***) = (** (v_1, l_1, u_1) < $\ldots (v_n, l_n, u_n)$ > < **DL=** *dl* > < **DU=** *du* > **)**

v_i is a value in the domain of one of the variables, and l_i and u_i are the lower and upper bounds on the number of variables assigned to v_i. The values of *dl* and *du* are the lower and upper bounds on the number of variables assigned to values outside of $\{v_1, \ldots, v_n\}$.

The GCC statement specifies one or more global cardinality constraints. A *global cardinality constraint* (GCC) is a constraint that consists of a set of variables $\{x_1, \ldots, x_n\}$ and for each value v in $D = \bigcup_{i=1,\ldots,n} Dom(x_i)$, 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.

For example, the constraint that is specified with the statements

```
var (x1-x6) = [1, 4];
gcc(x1-x6) = ((1, 1, 2) (2, 1, 3) (3, 1, 3) (4, 2, 3));
```

expresses that at least one but no more than two variables in $x_1, \ldots, x_6$ can have value 1, at least one and no more than three variables can have value 2 (or value 3), and at least two and no more than three variables can have value 4. For example, an assignment $x_1 = 1, x_2 = 1, x_3 = 2, x_4 = 3, x_5 = 4$, and $x_6 = 4$ satisfies the constraint.

If a global cardinality constraint has common lower or upper bounds for many of the values in D, the DL= and DU= options can be used to specify the common lower and upper bounds.

For example, the previous specification could also be written as

```
gcc(x1-x6) = ((1, 1, 2) (4, 2, 3) DL=1 DU=3);
```

You can also specify missing values for the lower and upper bounds. The values of *dl* and *du* are substituted as appropriate. The previous example can also be expressed as

```
gcc(x1-x6) = ((1, ., 2) (4, 2, .) DL=1 DU=3);
```

The following statements specify that each of the values in $\{1, \ldots, 9\}$ can be assigned to at most one of the variables $x_1, \ldots, x_9$:

```
var (x1-x9) = [1, 9];
gcc(x1-x9) = (DL=0 DU=1);
```

Note that the preceding global cardinality constraint is equivalent to the alldifferent constraint that is expressed as:

```
var (x1-x9) = [1, 9];
alldiff(x1-x9);
```

If you do not specify the DL= and DU= options, the default lower and upper bound for any value in D that does not appear in the (v, l, u) format is 0 and the number of variables in the constraint, respectively.

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 (x1-x9) = [0, 10];
gcc(x1-x9) = ((1, 0, 0) (3, 0, 0) (5, 0, 0) (7, 0, 0) (9, 0, 0));
```

If the variable list is empty, the GCC constraint applies to all the variables.

LEXICO Statement

LEXICO *lexicographic_ordering_constraint-1* < *. . . lexicographic_ordering_constraint-n* > **;**

LEXORDER *lexicographic_ordering_constraint-1* < *. . . lexicographic_ordering_constraint-n* > **;**

where *lexicographic_ordering_constraint* is specified in the form

((*variable-list-1***)** *order_type* **(***variable-list-2***))**

where *variable-list-1* and *variable-list-2* are variable lists of equal length. The keyword *order_type* signifies the type of ordering and can be one of two values: LEX_LE, which indicates lexicographically less than or equal to ($\leq_{\text{lex}}$), or LEX_LT, which indicates lexicographically less than ($<_{\text{lex}}$).

The LEXICO statement specifies one or more lexicographic constraints. The lexicographic constraint $\leq_{\text{lex}}$ and the strict lexicographic constraint $<_{\text{lex}}$ are defined as follows. 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

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

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

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

Frisch et al. (2002) introduced an optimal algorithm to establish generalized arc consistency (GAC) for the $\leq_{\text{lex}}$ constraint between two vectors of variables. 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 given subset of n-tuples, and $<_{\text{lex}}$ is a strict partial order on a given 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 statement as follows:

```
lexico( (x1-x6) lex_le (y1-y6) );
```

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 kind of symmetry that arises in CSPs with matrices of decision variables.

LINCON Statement

LINCON *linear_constraint-1* < . . . ,*linear_constraint-n* > **;**

LINEAR *linear_constraint-1* < . . . ,*linear_constraint-n* > **;**

where *linear_constraint* has the form

linear_expression-l type linear_expression-r

where *linear_expression* has the form

< **+|-** >*linear_term-1* < . . . , **(+|-)** *linear_term-n* >

where *linear_term* has the form

(*variable* **|** *number*< * *variable* >**)**

The keyword *type* can be one of the following:

<, <=, =, >=, >, <>, LT, LE, EQ, GE, GT, NE

The LINCON statement allows for a very general specification of linear constraints. In particular, it allows for specification of the following types of equality or inequality constraints:

$$\sum_{j=1}^{n} a_{ij} x_j \; \{\leq \mid < \mid = \mid \geq \mid > \mid \neq\} \; b_i \quad \text{for } i = 1, \ldots, m$$

For example, the constraint $4x_1 - 3x_2 = 5$ can be expressed as

```
var x1 x2;
lincon 4 * x1 - 3 * x2 = 5;
```

and the constraints

$$\begin{aligned} 10x_1 - x_2 &\geq 10 \\ x_1 + 5x_2 &\neq 15 \end{aligned}$$

can be expressed as

```
var x1 x2;
lincon 10 <= 10 * x1 - x2,
       x1 + 5 * x2 <> 15;
```

Note that variables can be specified on either side of an equality or inequality in a LINCON statement. Linear constraints can also be specified by using the CONDATA= data set.

Regardless of the specification, you must define the variables by using a VARIABLE statement or implicitly by specifying the USECONDATAVARS= option.

User-specified scalar values are subject to rounding based upon a platform-dependent tolerance.

OBJ Statement

OBJ *options* **;**

The OBJ statement enables you to set upper and lower bounds on the value of an objective function that is specified in the Constraint data set. You can also use the OBJ statement to specify the tolerance used for finding a locally optimal objective value.

If upper and lower bounds for the objective value are not specified, the CLP procedure tries to derive bounds from the domains of the variables that appear in the objective function. The procedure terminates with an error message if the objective is unbounded.

You can specify the following options in the OBJ statement:

LB=*m*

specifies the lower bound of the objective value.

TOL=*m*

specifies the tolerance of the objective value. The tolerance must not be less than 1E–6, which is the default.

UB=*m*

specifies the upper bound of the objective value.

For more information about using an objective function, see the section “Objective Function” on page 45.

PACK Statement

PACK *bin_packing_constraint-1* < . . . *bin_packing_constraint-n* > **;**

where *bin_packing_constraint* is specified by the form

$((b_1 < \ldots b_k >) (s_1 < \ldots s_k >) (l_1 < \ldots l_m >))$

The PACK constraint is used to assign k items to m bins, subject to the sizes of the items and the capacities of the bins. The item variable b_i assigns a bin to the ith item so that the domain for the item variables is 1 to m. Optionally, the domain for the item variables can be any contiguous set up to size m, where the smallest member represents the first bin, the second-smallest member represents the second bin, and so on. The constant s_i gives the size or weight of the ith item. The value of load variable l_j is determined by the total size of the contents of the corresponding bin.

For example, suppose there are three bins with capacities 3, 4, and 5. There are five items with sizes 4, 3, 2, 2, and 1 to be assigned to these three bins. The following statements formulate the problem and find all solutions:

```
proc clp out=out findallsolns;
   var bin1 = [0,3];
   var bin2 = [0,4];
   var bin3 = [0,5];
   var (item1-item5) = [1,3];
   pack ((item1-item5) (4,3,2,2,1) (bin1-bin3));
run;
```

Each row of Table 3.2 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 3.2 Bin Packing Solutions

	Item Variable				
	item1	**item2**	**item3**	**item4**	**item5**
	2	3	3	1	1
	2	3	1	3	1
	2	1	3	3	3
	3	1	2	2	3

NOTE: In specifying a PACK constraint, it can be more efficient to list the item variables in order by nonincreasing size and to specify VARSELECT=FIFO in the PROC CLP statement so that load variables are selected last.

REIFY Statement

REIFY *reify_constraint-1* < . . . *reify_constraint-n* > **;**

where *reify_constraint* is specified in the following form:

variable : constraint

The REIFY statement 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, the only type of constraint that can be reified is the linear constraint, which should have the same form as *linear_constraint* defined in the LINCON statement. As with the other variables, the control variable must also be defined in a VARIABLE statement or in the CONDATA= data set.

The REIFY statement 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 with the following statements:

```
var x y p q;
reify p: (3 * x + 4 * y < 20) q: (5 * x - 2 * y > 50);
lincon p + q >= 1;
```

The binary variables p and q reify the linear constraints

$$3x + 4y < 20$$

and

$$5x - 2y > 50$$

respectively. The following linear constraint enforces the desired disjunction:

$$p + q \geq 1$$

The implication constraint

$$(3x + 4y < 20) \Rightarrow (5x - 2y > 50)$$

can be enforced with the linear constraint

$$q - p \geq 0$$

The REIFY constraint can also be used to express a constraint that involves the absolute value of a variable. For example, the constraint

$$|X| = 5$$

can be expressed with the following statements:

```
var x p q;
reify p: (x = 5) q: (x = -5);
lincon p + q = 1;
```

REQUIRES Statement

REQUIRES *resource_constraint-1* < . . . *resource_constraint-n* > **;**

where *resource_constraint* is specified in the following form:

activity_specification = (resource_specification) < **QTY** = q >

where

activity_specification: (activity | activity-1 < . . . *activity-m* >*)*

and

resource_specification: (resource-1 < **QTY** = r_1 > < . . . **(, | OR)** *resource-l* < **QTY** = r_l > >*)*

activity_specification is a single activity or a list of activities that requires q units of the resource identified in *resource_specification*. *resource_specification* is a single resource or a list of resources, representing a choice of resource, along with the equivalent required quantities for each resource. The default value of r_i is 1. Alternate resource requirements are separated by a comma (,) or the keyword OR. The QTY= parameter outside the *resource_specification* acts as a multiplier to the QTY= parameters inside the *resource_specification*.

The REQUIRES statement defines the potential activity assignments with respect to the pool of resources. If an activity is not defined, the REQUIRES statement implicitly defines the activity.

You can also define resource constraints by using the Activity and Resource data sets in lieu of, or in conjunction with, the REQUIRES statement. Any resource constraints that are defined for an activity by using a REQUIRES statement override all resource constraints for that activity that are defined by using the Activity and Resource data sets.

The following statements illustrate how you would use a REQUIRES statement to specify that activity A requires resource R:

```
activity A;
resource R;
requires A = (R);
```

In order to specify that activity A requires two units of the resource R, you would add the QTY= keyword as in the following example:

```
requires A = (R qty=2);
```

In certain situations, the assignment might not be established in advance and there might be a set of possible alternates that can satisfy the requirements of an activity. This scenario can be defined by using multiple *resource-specifications* separated by commas or the keyword OR. For example, if the activity A needs either two units of the resource R1 or one unit of the resource R2, you could use the following statement:

```
requires A = (R1 qty=2, R2);
```

The equivalent statement using the keyword OR is

```
requires A = (R1 qty=2 or R2);
```

It is important to note that resources specified in a single resource constraint are disjunctive and not conjunctive. The activity is satisfied by exactly one of the resources rather than a combination of resources. For example, the following statement specifies that the possible resource assignment for activity A is either four units of R1 or two units of R2:

```
requires A = (R1 qty=2 or R2) qty=2;
```

The preceding statement does not, for example, result in an assignment of two units of the resource R1 and one unit of R2.

In order to model conjunctive resources by using a REQUIRES statement, such as when an activity requires more than one resource simultaneously, you need to define multiple resource constraints. For example, if activity A requires both resource R1 and resource R2, you can model it as follows:

```
requires A = (R1)  A = (R2);
```

or

```
requires A = (R1);
requires A = (R2);
```

If multiple activities have the same resource requirements, you can use an activity list for specifying the constraints instead of having separate constraints for each activity. For example, if activities A and B require resource R1 or resource R2, the specification

```
requires (A B) = (R1, R2);
```

is equivalent to

```
requires A = (R1, R2);
requires B = (R1, R2);
```

RESOURCE Statement

RESOURCE *resource_specification-1* < . . . *resource_specification-n* > **;**

where *resource_specification* is specified in the following form:

resource | *(resource-1* < . . . *resource-m* >*)* < =*(capacity)* >

The RESOURCE statement specifies the names and capacities of all resources that are available to be assigned to any defined activities. For example, the following statement specifies that there are two units of the resource R1 and one unit of the resource R2.

```
resource R1=(2) R2;
```

The capacity of a resource can take nonnegative integer values. The default capacity is 1, which corresponds to a unary resource.

SCHEDULE Statement

SCHEDULE *options* **;**

SCHED *options* **;**

The following options can appear in the SCHEDULE statement.

ACTASSIGN=*keyword*

specifies the activity assignment strategy subject to the activity selection strategy, which is specified by the ACTSELECT= option. After an activity has been selected, the activity assignment strategy determines a start time and a set of resources (if applicable based on resource requirements) for the selected activity.

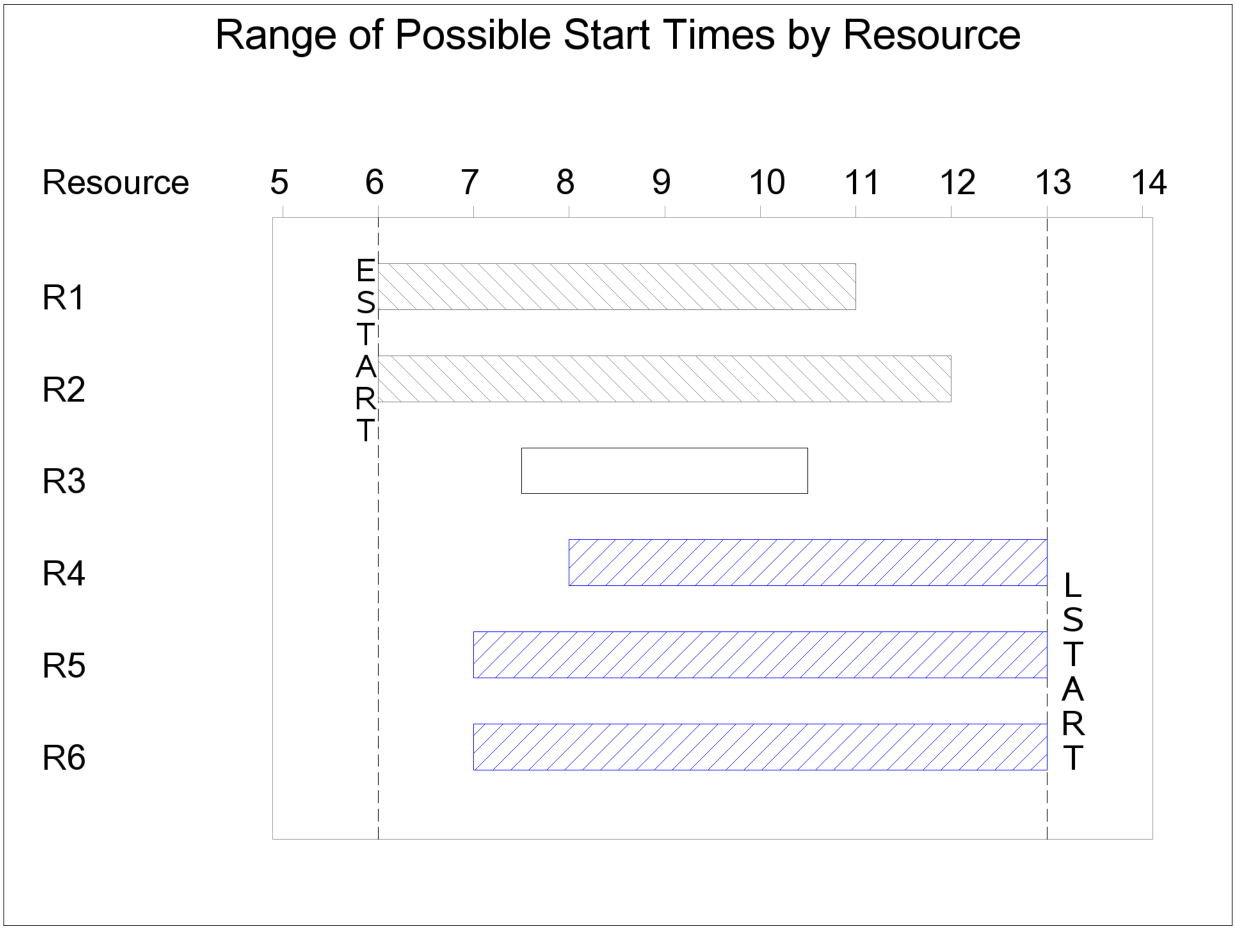
By default, an activity is assigned its earliest possible start time.

If an activity has any resource requirements, then the activity is assigned a set of resources as follows:

MAXTW	selects the set of resources that supports the assigned start time and affords the maximum time window of availability for the activity. Ties are broken randomly.
RAND	randomly selects a set of resources that support the selected start time for the activity.

For example, Figure 3.2 illustrates possible start times for a single activity which requires one of the resources R1, R2, R3, R4, R5, or R6. The bars depict the possible start times that are supported by each of the resources for the duration of the activity.

Figure 3.2 Range of Possible Start Times by Resource



Default behavior dictates that the activity is assigned its earliest possible start time of 6. Then, one of the resources that supports the selected start time (R1 and R2) is assigned. Specifically, if ACTASSIGN=RAND, the strategy randomly selects between R1 and R2. If ACTASSIGN=MAXTW, the strategy selects R2 because R1 has a smaller time window.

There is one exception to the preceding assignments. When ACTSELECT=RJRAND, an activity is assigned its latest possible start time. For the example in Figure 3.2, the activity is assigned its latest possible start time of 13 and one of R4, R5, or R6 is assigned. Specifically, if ACTASSIGN=RAND, the strategy randomly selects between R4, R5, and R6. If ACTASSIGN=MAXTW, the strategy randomly selects between R5 and R6 because their time windows are the same size (larger than the time window of R4).

The default activity assignment strategy is RAND. For assigning variables, use the VARASSIGN= option in the PROC CLP statement.

ACTSELECT=*keyword*

specifies the activity selection strategy. The activity selection strategy can be randomized or deterministic.

The following selection strategies use a random heuristic to break ties:

MAXD	selects an activity at random from those that begin prior to the earliest early finish time and that have maximum duration.
MINA	selects an activity at random from those that begin prior to the earliest early finish time and that have the minimum number of resource assignments.
MINLS	selects an activity at random from those that begin prior to the earliest early finish time and that have a minimum late start time.
PRIORITY	selects an activity at random from those that have the highest priority.
RAND	selects an activity at random from those that begin prior to the earliest early finish time. This strategy was proposed by Nuijten (1994).
RJRAND	selects an activity at random from those that finish after the latest late start time.

The following are deterministic selection strategies:

DET	selects the first activity that begins prior to the earliest activity finish time.
DMINLS	selects the activity with the earliest late start time.

The first activity is defined according to its appearance in the following order of precedence:

1. ACTIVITY statement
2. REQUIRES statement
3. ACTDATA= data set

The default activity selection strategy is RAND. For selecting variables, use the VARSELECT= option in the PROC CLP statement.

DURATION=*dur*

SCHEDDUR=*dur*

DUR=*dur*

specifies the duration of the schedule. The DURATION= option imposes a constraint that the duration of the schedule does not exceed the specified value.

EDGEFINDER *<=eftype>*

EDGE *<=eftype>*

activates the edge-finder consistency routines for scheduling CSPs. By default, the EDGEFINDER= option is inactive. Specifying the EDGEFINDER= option determines whether an activity must be the first or the last to be processed from a set of activities that require a given resource or set of resources and prunes the domain of the activity appropriately.

Valid values for the *eftype* keyword are FIRST, LAST, or BOTH. Note that *eftype* is an optional argument, and that specifying EDGEFINDER by itself is equivalent to specifying EDGEFINDER=LAST. The interpretation of each of these keywords is described as follows:

- FIRST: The edge-finder algorithm attempts to determine whether an activity must be processed first from a set of activities that require a given resource or set of resources and prunes its domain appropriately.
- LAST: The edge-finder algorithm attempts to determine whether an activity must be processed last from a set of activities that require a given resource or set of resources and prunes its domain appropriately.
- BOTH: This is equivalent to specifying both FIRST and LAST. The edge-finder algorithm attempts to determine which activities must be first and which activities must be last, and updates their domains as necessary.

There are several extensions to the edge-finder consistency routines. These extensions are invoked by using the NOTFIRST= and NOTLAST= options in the SCHEDULE statement. For more information about options that are related to edge-finder consistency routines, see the section "Edge Finding" on page 52.

EVALACTSEL*<=(keyword(s))>*

evaluates specified activity selection strategies by attempting to find a solution with each strategy. You can specify any combination of valid activity selection strategies in a space-delimited list enclosed in parentheses. If you do not specify a list, all available strategies are evaluated in alphabetical order, except that the default strategy is evaluated first. Descriptions of the available selection strategies are provided in the discussion of the ACTSELECT= option.

When the EVALACTSEL= option is in effect, the MAXTIME= option must also be specified. By default, the value specified for the MAXTIME= option is used as the maximum solution time for each activity selection strategy. When the DECRMAXTIME option is specified and a solution has been found, the value of the MAXTIME= option is set to the solution time of the last solution.

After the CLP procedure has attempted to find a solution with a particular strategy, it proceeds to the next strategy in the list. For this reason, the ACTSELECT=, ALLSOLNS, and MAXSOLNS= options are ignored when the EVALACTSEL= option is in effect. All solutions found during the evaluation process are saved in the output data set specified by the SCHEDULE= option.

The macro variable _ORCLPEAS_ provides an evaluation of each activity selection strategy. The fastest activity selection strategy is indicated in the macro variable _ORCLP_, provided at least one solution is found. See "Macro Variable _ORCLP_" on page 53 for more information about the _ORCLP_ macro variable; see "Macro Variable _ORCLPEAS_" on page 54 for more information about the _ORCLPEAS_ macro variable.

FINISH=*finish*

END=*finish*

FINISHBEFORE=*finish*

specifies the finish time for the schedule. The schedule finish time is an upper bound on the finish time of each activity (subject to time, precedence, and resource constraints). If you want to impose a tighter upper bound for an activity, you can do so either by using the FLE= specification in an ACTIVITY statement or by using the _ALIGNDATE_ and _ALIGNTYPE_ variables in the ACTDATA= data set.

NOTFIRST=*level*

NF=*level*

activates an extension of the edge-finder consistency routines for scheduling CSPs. By default, the NOTFIRST= option is inactive. Specifying the NOTFIRST= option determines whether an activity cannot be the first to be processed from a set of activities that require a given resource or set of resources and prunes its domain appropriately.

The argument *level* is numeric and indicates the level of propagation. Valid values are 1, 2, or 3, with a higher number reflecting more propagation. More propagation usually comes with a higher performance cost; the challenge is to strike the right balance. Specifying the NOTFIRST= option implicitly turns on the EDGEFINDER=LAST option because the latter is a special case of the former.

The corresponding NOTLAST= option determines whether an activity cannot be the last to be processed from a set of activities that require a given resource or set of resources.

For more information about options that are related to edge-finder consistency routines, see the section "Edge Finding" on page 52.

NOTLAST=*level*

NL=*level*

activates an extension of the edge-finder consistency routines for scheduling CSPs. By default, the NOTLAST= option is inactive. Specifying the NOTLAST= option determines whether an activity cannot be the last to be processed from a set of activities that require a given resource or set of resources and prunes its domain appropriately.

The argument *level* is numeric and indicates the level of propagation. Valid values are 1, 2, or 3, with a higher number reflecting more propagation. More propagation usually comes with a higher performance cost; the challenge is to strike the right balance. Specifying the NOTLAST= option implicitly turns on the EDGEFINDER=FIRST option because the latter is a special case of the former.

The corresponding NOTFIRST= option determines whether an activity cannot be the first to be processed from a set of activities requiring a given resource or set of resources.

For more information about options that are related to edge-finder consistency routines, see the section "Edge Finding" on page 52.

START=*start*

BEGIN=*start*

STARTAFTER=*start*

specifies the start time for the schedule. The schedule start time is a lower bound on the start time of each activity (subject to time, precedence, and resource constraints). If you want to impose a tighter lower bound for an activity, you can do so either by using the SGE= specification in an ACTIVITY statement or by using the _ALIGNDATE_ and _ALIGNTYPE_ variables in the ACTDATA= data set.

VARIABLE Statement

VARIABLE *var_specification-1* < . . . *var_specification-n* > **;**

VAR *var_specification-1* < . . . *var_specification-n* > **;**

A *var_specification* can be one of the following types:

variable < =[lower-bound < , upper-bound>]>

(variables) < =[lower-bound < , upper-bound>]>

The VARIABLE statement declares all variables that are to be considered in the CSP and, optionally, defines their domains. Any variable domains defined in a VARIABLE statement override the global variable domains that are defined by using the DOMAIN= option in the PROC CLP statement in addition to any bounds that are defined by using the CONDATA= data set. The values of *lower-bound* and *upper-bound* can also be specified as missing, in which case the appropriate values from the DOMAIN= specification are substituted.

Details: CLP Procedure

Type of Solvers

The CLP procedure features two different solvers. The choice of which one to use is determined by the nature of the CSP that you are attempting to solve.

- The *standard solver* is used to solve CSPs that are defined by alldifferent constraints, cumulative constraints, element constraints, GCC constraints, lexicographic constraints, linear constraints, pack constraints, reified constraints, ARRAY statements, and FOREACH statements. When you invoke the standard solver, the decision variables are one-dimensional; a variable is assigned an integer in a solution.

- The *scheduling solver* is used to solve CSPs that are defined by scheduling-specific constraints, such as temporal constraints (precedence and time) and resource constraints. When you use the scheduling solver, the variables are typically multidimensional; a variable is assigned a start time and possibly a set of resources in a solution. Furthermore, the variables are referred to as activities, and the solution is referred to as a schedule.

Selecting the Solver Type

The CLP procedure requires the specification of an output data set to store one or more solutions to the CSP. There are four possible output data sets: the Solution data set (specified using the OUT= option in the PROC CLP statement), which corresponds to the standard solver, and one or more Schedule data sets (specified using the SCHEDULE=, SCHEDRES=, or SCHEDTIME= options in the PROC CLP statement), which correspond to the scheduling solver. The solver type is determined by the type of output data set specified. If a Solution data set has been specified, the standard solver is invoked. If a Schedule data set has been specified, the scheduling solver is invoked. If an output data set is not specified, the procedure terminates with an error message. If both types of output data sets have been specified, the standard solver is invoked and the schedule-related data sets are ignored.

Constraint Data Set

The Constraint data set defines linear constraints, variable types, bounds on variable domains, and an objective function. You can use a Constraint data set in lieu of, or in combination with, a LINCON or a VARIABLE statement (or both) in order to define linear constraints, variable types, and variable bounds. You can use the Constraint data set in lieu of, or in combination with, the OBJ statement to specify an objective function. The Constraint data set is specified by using the CONDATA= option in the PROC CLP statement.

The Constraint data set must be in dense input format. In this format, a model's columns appear as variables in the input data set and the data set must contain the _TYPE_ variable, at least one numeric variable, and any reserved variables. Currently, the only reserved variable is the _RHS_ variable. If this requirement is not met, the CLP procedure terminates. The _TYPE_ variable is a character variable that tells the CLP procedure how to interpret each observation. The CLP procedure recognizes the following keywords as valid values for the _TYPE_ variable: EQ, LE, GE, NE, LT, GT, LOWERBD, UPPERBD, BINARY, FIXED, MAX, and MIN. An optional character variable, _ID_, can be used to name each row in the Constraint data set.

Linear Constraints

For the _TYPE_ values EQ, LE, GE, NE, LT, and GT, the corresponding observation is interpreted as a linear constraint. The _RHS_ variable is a numeric variable that contains the right-hand-side coefficient of the linear constraint. Any numeric variable other than _RHS_ that appears in a VARIABLE statement is interpreted as a structural variable for the linear constraint.

The _TYPE_ values are defined as follows:

EQ (=) defines a linear equality of the form

$$\sum_{j=1}^{n} a_{ij} x_j = b_i$$

LE (<=) defines a linear inequality of the form

$$\sum_{j=1}^{n} a_{ij} x_j \le b_i$$

GE (>=) defines a linear inequality of the form

$$\sum_{j=1}^{n} a_{ij} x_j \ge b_i$$

NE (<>) defines a linear inequality of the form

$$\sum_{j=1}^{n} a_{ij} x_j \ne b_i$$

LT (<) defines a linear inequality of the form

$$\sum_{j=1}^{n} a_{ij} x_j < b_i$$

GT (>) defines a linear inequality of the form

$$\sum_{j=1}^{n} a_{ij} x_j > b_i$$

Domain Bounds

The keywords LOWERBD and UPPERBD specify additional lower bounds and upper bounds, respectively, on the variable domains. In an observation where the _TYPE_ variable is equal to LOWERBD, a non-missing value for a decision variable is considered to be a lower bound for that variable. Similarly, in an observation where the _TYPE_ variable is equal to UPPERBD, a non-missing value for a decision variable is considered to be an upper bound for that variable. Note that lower and upper bounds defined in the Constraint data set are overridden by lower and upper bounds that are defined by using a VARIABLE statement.

Variable Types

The keywords BINARY and FIXED specify numeric variable types. If the value of _TYPE_ is BINARY for an observation, then any decision variable with a non-missing entry for the observation is interpreted as being a binary variable with domain $\{0,1\}$. If the value of _TYPE_ is FIXED for an observation, then any decision variable with a non-missing entry for the observation is interpreted as being assigned to that non-missing value. In other words, if the value of the variable X is c in an observation for which _TYPE_ is FIXED, then the domain of X is considered to be the singleton $\{c\}$. The value c should belong to the domain of X, or the problem is deemed infeasible.

Objective Function

The keywords MAX and MIN specify the objective function of a maximization or a minimization problem, respectively. In an observation where the _TYPE_ variable is equal to MAX or MIN, a non-missing value for a decision variable is the coefficient of this variable in the objective function. The value specified for _RHS_ is ignored in this case.

The bisection method is used to find the optimal objective value within the specified or derived lower and upper bounds. A solution is considered optimal if the difference between consecutive objective values is less than or equal to the tolerance. You can use the OBJ statement to specify the tolerance in addition to upper and lower bounds on the objective value. The minimum objective tolerance is 1E–6, which is the default.

When an optimal solution is found, the solution is stored in the output data set and the resulting objective value is stored in the macro variable _ORCLP_. The objective value is not necessarily optimal when it is computed within a time limit specified by the MAXTIME= option. In this case, the last valid solution computed within the time limit appears in the output data set. See the macro variable _ORCLP_ for more information about solution status.

The MAX and MIN functions are defined as follows:

MAX defines an objective function of the form

$$\max \sum_{j=1}^{n} c_j x_j$$

MIN defines an objective function of the form

$$\min \sum_{j=1}^{n} c_j x_j$$

Variables in the CONDATA= Data Table

Table 3.3 lists all the variables that are associated with the Constraint data set and their interpretations by the CLP procedure. For each variable, the table also lists its type (C for character, N for numeric), the possible values it can assume, and its default value.

Table 3.3 Constraint Data Table Variables

Name	Type	Description	Allowed Values	Default
TYPE	C	Observation type	EQ, LE, GE, NE, LT, GT, LOWERBD, UPPERBD, BINARY, FIXED, MAX, MIN	
RHS	N	Right-hand-side coefficient		0
_ID_	C	Observation name (optional)		
Any numeric variable other than _RHS_	N	Structural variable		

Solution Data Set

In order to solve a standard (nonscheduling) type CSP, you need to specify a Solution data set by using the OUT= option in the PROC CLP statement. The Solution data set contains all the solutions that have been determined by the CLP procedure. You can specify an upper bound on the number of solutions by using the MAXSOLNS= option in the PROC CLP statement. If you prefer that PROC CLP determine all possible solutions instead, you can specify the FINDALL option in the PROC CLP statement.

The Solution data set contains as many decision variables as have been defined in the CLP procedure invocation. Every observation in the Solution data set corresponds to a solution to the CSP. If a Constraint data set has been specified, then any variable formats and variable labels from the Constraint data set carry over to the Solution data set.

Activity Data Set

You can use an Activity data set in lieu of, or in combination with, an ACTIVITY statement to define activities and constraints that relate to the activities. The Activity data set is similar to the Activity data set of the CPM procedure in SAS/OR software and is specified by using the ACTDATA= option in the PROC CLP statement.

The Activity data set enables you to define an activity, its domain, temporal constraints, resource constraints, and priority. The temporal constraints can be either time-alignment-type or precedence-type constraints. The Activity data set requires at least two variables: one to determine the activity, and another to determine its duration. The procedure terminates if it cannot find the required variables. The activity is determined with the _ACTIVITY_ variable, which must be character, and the duration is determined with the _DURATION_ variable, which must be numeric. You can define temporal constraints, resource constraints, and priority by including additional variables.

Time Alignment Constraints

The _ALIGNDATE_ and _ALIGNTYPE_ variables enable you to define time-alignment-type constraints. The _ALIGNTYPE_ variable defines the type of the alignment constraint for the activity that is named in the _ACTIVITY_ variable with respect to the _ALIGNDATE_ variable. If the _ALIGNDATE_ variable is not present in the Activity data set, the _ALIGNTYPE_ variable is ignored. Similarly, _ALIGNDATE_ is ignored when _ALIGNTYPE_ is not present. The _ALIGNDATE_ variable can take nonnegative integer values. The _ALIGNTYPE_ variable can take the values shown in Table 3.4.

Table 3.4 Valid Values for the _ALIGNTYPE_ Variable

Value	Type of Alignment
SEQ	Start equal to
SGE	Start greater than or equal to
SLE	Start less than or equal to
FEQ	Finish equal to
FGE	Finish greater than or equal to
FLE	Finish less than or equal to

Precedence Constraints

The _SUCCESSOR_ variable enables you to define precedence-type relationships between activities by using AON (activity-on-node) format. The _SUCCESSOR_ variable is a character variable. The _LAG_ variable defines the lag type of the relationship. By default, all precedence relationships are considered to be *finish-to-start* (*FS*). An FS type of precedence relationship is also referred to as a *standard* precedence constraint. All other types of precedence relationships are considered to be nonstandard precedence constraints. The _LAGDUR_ variable specifies the lag duration. By default, the lag duration is zero.

For each (activity, successor) pair, you can define a lag type and a lag duration. Consider a pair of activities (A, B) with a lag duration represented by *lagdur* in Table 3.5. The interpretation of each of the different lag types is given in Table 3.5.

Table 3.5 Valid Values for the _LAG_ Variable

Lag Type	Interpretation
FS	Finish A + lagdur $\leq$ Start B
SS	Start A + lagdur $\leq$ Start B
FF	Finish A + lagdur $\leq$ Finish B
SF	Start A + lagdur $\leq$ Finish B
FSE	Finish A + lagdur = Start B
SSE	Start A + lagdur = Start B
FFE	Finish A + lagdur = Finish B
SFE	Start A + lagdur = Finish B

The first four lag types (FS, SS, FF, and SF) are also referred to as *finish-to-start*, *start-to-start*, *finish-to-finish*, and *start-to-finish*, respectively. The next four types (FSE, SSE, FFE, and SFE) are stricter versions of FS, SS, FF, and SF, respectively. The first four types impose a lower bound on the start and finish times of B, while the last four types force the start and finish times to be set equal to the lower bound of the domain. The last four types enable you to force an activity to begin when its predecessor is finished. It is relatively easy to generate infeasible scenarios with the stricter versions, so you should use the stricter versions only if the weaker versions are not adequate for your problem.

Resource Constraints

The _RESOURCE_ and _QTY_ variables enable you to define resource constraints for activities. The _RESOURCE_ variable is a character variable that identifies the resource or resource pool. The _QTY_ variable is a numeric variable that identifies the number of units required. If the requirement is for a resource pool, you need to use the Resource data set to identify the pool members. See the section "Resource Data Set" on page 49 for more information.

For example, the following observations specify that activity A1 needs one unit of resource R1 and two units of resource R2:

```
_ACTIVITY_     _RESOURCE_     _QTY_

    A1             R1           1
    A1             R2           2
```

Activity Priority

The _PRIORITY_ variable enables you to specify an activity's priority for use with the PRIORITY selection strategy of the ACTSELECT= option. The _PRIORITY_ variable can take any integer value. Lower numbers indicate higher priorities; a missing value is treated as $+\infty$. If the ACTSELECT=PRIORITY option is specified without the _PRIORITY_ variable, all activities are assumed to have equal priorities.

Variables in the ACTDATA= Data Set

Table 3.6 lists all the variables that are associated with the ACTDATA= data set and their interpretations by the CLP procedure. For each variable, the table also lists its type (C for character, N for numeric), its possible values, and its default value.

Table 3.6 Activity Data Set Variables

Name	Type	Description	Allowed Values	Default
ACTIVITY	C	Activity name		
DURATION	N	Duration	Nonnegative integers	0
SUCCESSOR	C	Successor name		
LAG	C	Lag type	FS, SS, FF, SF, FSE, SSE, FFE, SFE	FS
LAGDUR	N	Lag duration		0
ALIGNDATE	N	Alignment date		
ALIGNTYPE	C	Alignment type	SGE, SLE, SEQ, FGE, FLE, FEQ	
RESOURCE	C	Resource name		
QTY	N	Resource quantity	Nonnegative integers	1
PRIORITY	N	Activity priority	Integers	$+\infty$

Resource Data Set

The Resource data set is used in conjunction with the ACTDATA= data set to define resources, resource capacities, and alternate resources. The Resource data set contains at most four variables: _RESOURCE_, _CAPACITY_, _POOL_, and _SUBQTY_. The Resource data set is specified by using the RESDATA= option in the PROC CLP statement.

The _RESOURCE_ variable is a required character variable that defines resources. The _CAPACITY_ variable is a numeric variable that defines the capacity of the resource; it takes only nonnegative integer values. In the absence of alternate resources, the _RESOURCE_ and _CAPACITY_ variables are the only variables that you need in a data set to define resources and their capacities.

The following Resource data set defines resource R1 with capacity 2 and resource R2 with capacity 4:

```
_RESOURCE_    _CAPACITY_

    R1              2
    R2              4
```

Now suppose that you have an activity whose resource requirements can be satisfied by any one of a given set of resources. The Activity data set does not directly allow for a disjunctive specification. In order to provide a disjunctive specification, you need to specify an abstract resource, referred to as a *resource pool*, in the _RESOURCE_ variable and use the _POOL_ and _SUBQTY_ variables in the Resource data set to identify the resources that can be substituted for this resource pool. The _POOL_ variable is a character variable that identifies a resource pool to which the _RESOURCE_ variable belongs. The _SUBQTY_ variable is a numeric variable that identifies the number of units of _RESOURCE_ that can substitute for one unit of the resource pool. The _SUBQTY_ variable takes only nonnegative integer values. Each resource pool corresponds to as many observations in the Resource data set as there are members in the pool. A _RESOURCE_ can have membership in more than one resource pool. The resource and resource pool are distinct entities in the Resource data set; that is, a _RESOURCE_ cannot have the same name as a _POOL_ in the Resource data set and vice versa.

For example, consider the following Activity data set:

```
Obs    _ACTIVITY_    _DURATION_    _RESOURCE_

 1         A              1            R1
 2         B              2            RP1
 3         C              1            RP2
```

and Resource data set:

```
Obs    _RESOURCE_    _CAPACITY_    _POOL_    _SUBQTY_

 1         R1             2          RP1         1
 2         R2             1          RP1         1
 3         R1             2          RP2         2
 4         R2             1          RP2         1
```

Activity A requires the resource R1. Activity B requires the resource RP1, which is identified as a resource pool in the Resource data set with members R1 and R2. Since the value of _SUBQTY_ is 1 for both resources, activity B can be satisfied with one unit of R1 or one unit of R2. Observations 3 and 4 in the Resource data set define resource pool RP2. Activity C requires resource pool RP2, which translates to requiring two units of R1 or one unit of R2 (since the value of _SUBQTY_ is 2 in observation 3 of the Resource data set). Resource substitution is not a shareable substitution; it is all or nothing. For example, if activity A requires two units of RP1 instead, the substitution is two units of R1 or two units of R2. The requirement cannot be satisfied using one unit of R1 and one unit of R2.

Variables in the RESDATA= Data Set

Table 3.7 lists all the variables that are associated with the RESDATA= data set and their interpretations by the CLP procedure. For each variable, the table also lists its type (C for character, N for numeric), its possible values, and its default value.

Table 3.7 Resource Data Set Variables

Name	Type	Description	Allowed Values	Default
RESOURCE	C	Resource name		
CAPACITY	N	Resource capacity	Nonnegative integers	1
POOL	C	Resource pool name		
SUBQTY	N	Number of units of resource that can substitute for one unit of the resource pool	Nonnegative integers	1

Schedule Data Set

In order to solve a scheduling type CSP, you need to specify one or more schedule-related output data sets by using one or more of the SCHEDULE=, SCHEDTIME=, or SCHEDRES= options in the PROC CLP statement.

The Schedule data set is specified with the SCHEDULE= option in the PROC CLP statement and is the only data set that contains both time and resource assignment information for each activity.

The SCHEDULE= data set always contains the following five variables: SOLUTION, ACTIVITY, DURATION, START, and FINISH. The SOLUTION variable gives the solution number to which each observation corresponds. The ACTIVITY variable identifies each activity. The DURATION variable gives the duration of the activity. The START and FINISH variables give the scheduled start and finish times for the activity. There is one observation that contains the time assignment information for each activity that corresponds to these variables.

If any resources have been specified, the data set contains three more variables: OBSTYPE, RESOURCE, and QTY. The value of the OBSTYPE variable indicates whether an observation represents time assignment information or resource assignment information. Observations that correspond to OBSTYPE="TIME" provide time assignment information, and observations that correspond to OBSTYPE="RESOURCE" provide resource assignment information. The RESOURCE variable and the QTY variable constitute the resource assignment information and identify the resource and quantity, respectively, of the resource that is assigned to each activity.

The values of RESOURCE and QTY are missing for time assignment observations, and the values of DURATION, START, and FINISH are missing for resource assignment observations.

If an Activity data set has been specified, the formats and labels for the _ACTIVITY_ and _DURATION_ variables carry over to the ACTIVITY and DURATION variables, respectively, in the Schedule data set.

In addition to or in lieu of the SCHEDULE= data set, there are two other schedule-related data sets that together represent a logical partitioning of the Schedule data set with no loss of data. The SCHEDTIME= data set contains the time assignment information, and the SCHEDRES= data set contains the resource assignment information.

Variables in the SCHEDULE= Data Set

Table 3.8 lists all the variables that are associated with the SCHEDULE= data set and their interpretations by the CLP procedure. For each variable, the table also lists its type (C for character, N for numeric), and its possible values.

Table 3.8 Schedule Data Set Variables

Name	Type	Description	Values
SOLUTION	N	Solution number	Positive integers
OBSTYPE	C	Observation type	TIME, RESOURCE
ACTIVITY	C	Activity name	
DURATION	N	Duration	Nonnegative integers, missing when OBSTYPE="RESOURCE"
START	N	Start time	Missing when OBSTYPE="RESOURCE"
FINISH	N	Finish time	Missing when OBSTYPE="RESOURCE"
RESOURCE	C	Resource name	Missing when OBSTYPE="TIME"
QTY	N	Resource quantity	Nonnegative integers, missing when OBSTYPE="TIME"

SCHEDRES= Data Set

The SCHEDRES= data set contains the resource assignments for each activity. There are four variables: SOLUTION, ACTIVITY, RESOURCE, and QTY, which are identical to the same variables in the SCHEDULE= data set. The observations correspond to the subset of observations in the SCHEDULE= data set with OBSTYPE="RESOURCE."

SCHEDTIME= Data Set

The SCHEDTIME= data set contains the time assignments for each activity. There are five variables: SOLUTION, ACTIVITY, DURATION, START, and FINISH, which are identical to the same variables in the SCHEDULE= data set. The observations correspond to the subset of observations in the SCHEDULE= data set with OBSTYPE="TIME."

Edge Finding

Edge-finding (EF) techniques are effective propagation techniques for resource capacity constraints that reason about the processing order of a set of activities that require a given resource or set of resources. Some of the typical ordering relationships that EF techniques can determine are whether an activity can, cannot, or must execute before (or after) a set of activities that require the same resource or set of resources. This in turn determines new time bounds on the start and finish times. Carlier and Pinson (1989) are responsible for some of the earliest work in this area, which resulted in solving MT10, a 10×10 job shop problem that had remained unsolved for over 20 years (Muth and Thompson 1963). Since then, there have been several variations and extensions of this work (Carlier and Pinson 1990; Applegate and Cook 1991; Nuijten 1994; Baptiste and Le Pape 1996).

You invoke the edge-finding consistency routines by specifying the EDGEFINDER= or EDGE= option in the SCHEDULE statement. Specifying EDGEFINDER=FIRST computes an upper bound on the activity finish time by detecting whether a given activity must be processed first from a set of activities that require the same resource or set of resources. Specifying EDGEFINDER=LAST computes a lower bound on the activity start time by detecting whether a given activity must be processed last from a set of activities that require the same resource or set of resources. Specifying EDGEFINDER=BOTH is equivalent to specifying both EDGEFINDER=FIRST and EDGEFINDER=LAST.

An extension of the edge-finding consistency routines is determining whether an activity *cannot* be the first to be processed or whether an activity *cannot* be the last to be processed from a given set of activities that require the same resource or set of resources. The NOTFIRST= or NF= option in the SCHEDULE statement determines whether an activity must not be the first to be processed. In similar fashion, the NOTLAST= or NL= option in the SCHEDULE statement determines whether an activity must not be the last to be processed.

Macro Variable _ORCLP_

The CLP procedure defines a macro variable named _ORCLP_. This variable contains a character string that indicates the status of the CLP procedure upon termination. The various terms of the macro variable are interpreted as follows.

STATUS

indicates the procedure status at termination. It can take one of the following values:

OK	The procedure terminated successfully.
DATA_ERROR	An input data error occurred.
IO_ERROR	A problem in reading or writing data occurred.
MEMORY_ERROR	Insufficient memory is allocated to the procedure.
SEMANTIC_ERROR	The use of semantic action is incorrect.
SYNTAX_ERROR	The use of syntax is incorrect.
ERROR	The status cannot be classified into any of the preceding categories.

If PROC CLP terminates normally or if an I/O error is detected while the procedure is closing a data set, the following terms are added to the macro variable.

SOLUTION_STATUS

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

ALL_SOLUTIONS	All solutions are found.
INFEASIBLE	The problem is infeasible.
OPTIMAL	The solution is optimal.
SOLN_LIMIT_REACHED	The required number of solutions specified with the MAXSOLNS= option is reached.
TIME_LIMIT_REACHED	The execution time limit specified with the MAXTIME= option is reached.
RESTART_LIMIT_REACHED	The number of restarts specified with the RESTARTS= option is reached.
EVALUATION_COMPLETE	The evaluation process is finished. This solution status appears only when the EVALACTSEL= option or the EVALVARSEL= option is specified.
ABORT	The procedure is stopped by the user before any other stopping criterion is reached.

SOLUTIONS_FOUND

indicates the number of solutions that are found. This term is not applicable if SOLUTION_STATUS=INFEASIBLE.

MIN_MAKESPAN
indicates the minimal makespan of the solutions that are found. The makespan is the maximum of the activity finish times or the completion time of the last job to leave the system. This term is applicable only to scheduling problems that have at least one solution.

SOLUTION_TIME
indicates the time taken to solve the problem. By default, the time reported is CPU time; see the TIMETYPE= option for more information.

VARSELTYPE
indicates the fastest variable selection strategy. This term appears only when the EVALVARSEL= option is active and either at least one solution is found or the problem is found to be infeasible.

ACTSELTYPE
indicates the fastest activity selection strategy. This term appears only when the EVALACTSEL= option is active and at least one solution is found.

OBJECTIVE
indicates the objective value. This term appears only when an objective function is specified and at least one solution is found.

Macro Variable _ORCLPEAS_

When you specify the EVALACTSEL= option to evaluate activity selection strategies for a scheduling problem, the CLP procedure defines a macro variable named _ORCLPEAS_. This variable contains a character string that describes the solution attempt made with each selection strategy. The macro variable contains four terms for each selection strategy that is evaluated; these terms are interpreted as follows:

ACTSELTYPE
indicates the activity selection strategy being evaluated.

EVAL_TIME
indicates the amount of time taken to evaluate the activity selection strategy. By default, the time reported is CPU time; see the TIMETYPE= option for more information.

SOLUTION
indicates the index of the solution in the output data set, provided that a solution has been found. Otherwise, SOLUTION=NOT_FOUND.

REASON
indicates the reason a solution was not found. The reason can be either TIME_LIMIT_REACHED or RESTART_LIMIT_REACHED. This term is included only when SOLUTION=NOT_FOUND.

MAX_ACTSEL
indicates the maximum number of activities selected within the evaluation time.

Macro Variable _ORCLPEVS_

When you specify the EVALVARSEL= option to evaluate variable selection strategies for a standard CSP, the CLP procedure defines a macro variable named _ORCLPEVS_. This variable contains a character string that describes the solution attempt made with each selection strategy. The macro variable contains four terms for each selection strategy that is evaluated as follows:

VARSELTYPE
indicates the variable selection strategy being evaluated.

EVAL_TIME
indicates the amount of time taken to evaluate the variable selection strategy. By default, the time reported is CPU time; see the TIMETYPE= option for more information.

SOLUTION
indicates the index of the solution if a solution is found. Otherwise, SOLUTION=INFEASIBLE if the problem is found to be infeasible, or SOLUTION=NOT_FOUND.

MAX_VARSEL
indicates the maximum number of variables selected within the evaluation time.

Examples: CLP Procedure

This section contains several examples that illustrate the capabilities of the different logical constraints and showcase a variety of problems that the CLP procedure can solve. The following examples feature a standard constraint satisfaction problem (CSP):

- "Example 3.1: Logic-Based Puzzles" illustrates the capabilities of the alldifferent constraint in solving a popular logical puzzle, the sudoku. This example also contains a variant of sudoku that illustrates the capabilities of the GCC constraint. Finally, a Magic Square puzzle demonstrates the use of the EVALVARSEL= option.
- "Example 3.2: Alphabet Blocks Problem" illustrates how to use the GCC constraint to solve the alphabet blocks problem, a popular combinatorial problem.
- "Example 3.3: Work-Shift Scheduling Problem" illustrates the capabilities of the element constraint in modeling the cost information in a work-shift scheduling problem in order to find a minimum cost schedule.
- "Example 3.4: A Nonlinear Optimization Problem" illustrates how to use the element constraint to represent nonlinear functions and nonstandard variable domains, including noncontiguous domains. This example also demonstrates how to specify an objective function in the Constraint data set.
- "Example 3.5: Car Painting Problem" involves limited sequencing of cars in an assembly process in order to minimize the number of paint purgings; it features the reified constraint.

- "Example 3.6: Scene Allocation Problem" illustrates how to schedule the shooting of different movie scenes in order to minimize production costs. This problem uses the GCC and linear constraints.
- "Example 3.7: Car Sequencing Problem" relates to sequencing the cars on an assembly line with workstations for installing specific options subject to the demand constraints for each set of options and the capacity constraints of each workstation.
- "Example 3.13: Balanced Incomplete Block Design" illustrates how to use the lexicographic constraint to break symmetries in generating a balanced incomplete block design (BIBD), a standard combinatorial problem from design theory.
- "Example 3.14: Progressive Party Problem" illustrates how to use the pack constraint to solve the progressive party problem, a well-known constraint programming problem in which crews of various sizes must be assigned to boats of various capacities for several rounds of parties.
- "Example 3.15: Resource-Constrained Project Scheduling Problem with Time Windows" illustrates how to use the cumulative constraint to solve a resource-constrained project scheduling problem with time windows.
- "Example 3.16: Resource-Constrained Scheduling Problem with Nonstandard Temporal Constraints" illustrates how to use the cumulative constraint to solve a resource-constrained scheduling problem that has nonstandard temporal constraints.

The following examples feature scheduling CSPs and use the scheduling constraints in the CLP procedure:

- "Example 3.8: Round-Robin Problem" illustrates solving a single round-robin tournament.
- "Example 3.9: Resource-Constrained Scheduling with Nonstandard Temporal Constraints" illustrates nonstandard precedence constraints in scheduling the construction of a bridge.
- "Example 3.10: Scheduling with Alternate Resources" illustrates a job-scheduling problem with alternate resources. An optimal solution is determined by activating the edge-finding consistency techniques for this example.
- "Example 3.11: 10×10 Job Shop Scheduling Problem" illustrates a well-known 10×10 job shop scheduling problem and features edge-finding along with the edge-finding extensions "not first" and "not last" in order to determine optimality.

It is often possible to formulate a problem both as a standard CSP and also as a scheduling CSP. Depending on the nature of the constraints, it might even be more advantageous to formulate a scheduling problem as a standard CSP and vice versa:

- "Example 3.12: Scheduling a Major Basketball Conference" illustrates this concept by modeling the problem of scheduling a major basketball conference as a standard CSP. The element constraint plays a key role in this particular example.

Example 3.1: Logic-Based Puzzles

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

Sudoku

A 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 contain only one of each digit. Output 3.1.1 shows an example of a sudoku grid.

Output 3.1.1 An 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 ALLDIFFERENT constraint to solve the preceding sudoku problem.

The data set indata contains the partially filled values for the grid and is used to create the set of macro variables C_{ij} $(i = 1 \ldots 9, j = 1 \ldots 9)$, where C_{ij} is the value of cell (i, j) in the grid when specified, and missing otherwise.

```
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 . .
;

%macro store_initial_values;
   /* store initial values into macro variable C_i_j */
   data _null_;
      set indata;
      array C{9};
      do j = 1 to 9;
         i = _N_;
         call symput(compress('C_'||put(i,best.)||'_'||put(j,best.)),
```

```
            put(C[j],best.));
      end;
   run;
%mend store_initial_values;

%store_initial_values;
```

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]. Three sets of alldifferent constraints are used to set the required rules for each row, each column, and each of the 3×3 blocks. The constraint ALLDIFF($X_{i1} - X_{i9}$) forces all values in row *i* to be different, the constraint ALLDIFF($X_{1j} - X_{9j}$) forces all values in column *j* to be different, and the constraint ALLDIFF(X_{ij}) $((i = 1, 2, 3; j = 1, 2, 3), (i = 1, 2, 3; j = 4, 5, 6), \ldots, (i = 7, 8, 9; j = 7, 8, 9))$ forces all values in each block to be different.

The following statements solve the sudoku puzzle:

```
%macro solve;
   proc clp out=outdata;

      /* Declare variables    */
      /* Nine row constraints */
      %do i = 1 %to 9;
         var (X_&i._1-X_&i._9) = [1,9];
         alldiff(X_&i._1-X_&i._9);
      %end;

      /* Nine column constraints */
      %do j = 1 %to 9;
         alldiff(
         %do i = 1 %to 9;
            X_&i._&j
         %end;
         );
      %end;

      /* Nine 3x3 block constraints */
      %do s = 0 %to 2;
         %do t = 0 %to 2;
            alldiff(
            %do i = 3*&s + 1 %to 3*&s + 3;
               %do j = 3*&t + 1 %to 3*&t + 3;
                  X_&i._&j
               %end;
            %end;
            );
         %end;
      %end;

      /* Initialize variables to cell values   */
      /* X_i_j = C_i_j if C_i_j is non-missing */
      %do i = 1 %to 9;
         %do j = 1 %to 9;
            %if &&C_&i._&j ne . %then %do;
               lincon X_&i._&j = &&C_&i._&j;
            %end;
         %end;
      %end;

   run;
```

```
   %put &_ORCLP_;
%mend solve;

%solve
```

Output 3.1.2 shows the solution.

Output 3.1.2 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 π that occurs every March 14. In honor of Pi Day, Brainfreeze Puzzles (Riley and Taalman 2008) celebrated this day with a special 12×12 grid sudoku puzzle. The 2008 Pi Day sudoku puzzle is shown in Output 3.1.3.

Output 3.1.3 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 3×3 blocks, this puzzle uses jigsaw regions such that highlighted regions in the middle resemble the Greek letter π. Each jigsaw region consists of 12 contiguous cells.

2. The first 12 digits of π are used instead of the digits 1–9. Each row, column, and jigsaw region contains the first 12 digits of π (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.

The data set raw contains the partially filled values for the grid and, similar to the sudoku problem, is used to create the set of macro variables C_{ij} $(i = 1, \ldots, 12, j = 1, \ldots, 12)$ where C_{ij} is the value of cell (i, j) in the grid when specified, and missing otherwise.

```
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
;
```

```
%macro cdata;
   /* store each pre-filled value into macro variable C_i_j */
   data _null_;
      set raw;
      array C{12};
      do j = 1 to 12;
         i = _N_;
         call symput(compress('C_'||put(i,best.)||'_'||put(j,best.)),
            put(C[j],best.));
      end;
   run;
%mend cdata;
%cdata;
```

As in the sudoku problem, let the variable X_{ij} represent the value of the cell that corresponds to row i and column j. The domain of each of these variables is [1, 9].

For each row, column, and jigsaw region, a GCC statement is specified to enforce the condition that it contain exactly the first twelve digits of π.

In particular, the variables in row r, $r = 1, \ldots, 12$ are $X_{r1}, \ldots, X_{r12}$. The SAS macro %CONS_ROW(R) enforces the GCC constraint that row r contain exactly two 1s, two 3s, three 5s, no 7s, and one of each of the other values:

```
%macro cons_row(r);
   /* Row r 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.           */
   gcc(X_&r._1-X_&r._12) =
      ( (1, 2, 2) (3, 2, 2) (5, 3, 3) (7, 0, 0) DL=1 DU=1 );
%mend cons_row;
```

The variables in column c are $X_{1c}, \ldots, X_{12c}$. The SAS macro %CONS_COL(C) enforces a similar GCC constraint for each column c.

```
%macro cons_col(c);
   /* Column c 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. */
   gcc( %do r = 1 %to 12;
      X_&r._&c.
      %end;
      ) = ((1, 2, 2) (3, 2, 2) (5, 3, 3) (7, 0, 0) DL=1 DU=1);
%mend cons_col;
```

Generalizing this concept further, the SAS macro %CONS_REGION(VARS) enforces the GCC constraint for the jigsaw region that is defined by the macro variable VARS.

```
%macro cons_region(vars);
   /* Jigsaw region that contains &vars 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.                                     */
   gcc(&vars.) = ((1, 2, 2) (3, 2, 2) (5, 3, 3) (7, 0, 0) DL=1 DU=1);
%mend cons_region;
```

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

```
%macro pds(solns=allsolns,varsel=MINR,maxt=900);

   proc clp out=pdsout &solns
      varselect=&varsel /* Variable selection strategy */
      maxtime=&maxt;    /* Time limit                  */

      /* Variable X_i_j represents the grid of ith row and jth column. */
      var (
         %do i = 1 %to 12;
            X_&i._1 - X_&i._12
         %end;
         ) = [1,9];

      /* X_i_j = C_i_j if C_i_j is non-missing */
      %do i = 1 %to 12;
         %do j = 1 %to 12;
            %if &&C_&i._&j ne . %then %do;
               lincon X_&i._&j = &&C_&i._&j;
            %end;
         %end;
      %end;

      /* 12 Row constraints: */
      %do r = 1 %to 12;
         %cons_row(&r);
      %end;

      /* 12 Column constraints: */
      %do c = 1 %to 12;
         %cons_col(&c);
      %end;

      /* 12 Jigsaw region constraints: */
      /* Each jigsaw region is defined by the macro variable &vars. */

      /* Region 1: */
      %let vars = X_1_1 - X_1_3 X_2_1 - X_2_3
                  X_3_1 X_3_2 X_4_1 X_4_2 X_5_1 X_5_2;
      %cons_region(&vars.);

      /* Region 2: */
      %let vars = X_1_4 - X_1_9 X_2_4 - X_2_9;
      %cons_region(&vars.);

      /* Region 3: */
      %let vars = X_1_10 - X_1_12 X_2_10 - X_2_12
                  X_3_11 X_3_12 X_4_11 X_4_12 X_5_11 X_5_12;
      %cons_region(&vars.);

      /* Region 4: */
      %let vars = X_3_3 - X_3_6 X_4_3 - X_4_6 X_5_3 - X_5_6;
      %cons_region(&vars.);
```

```
      /* Region 5: */
      %let vars = X_3_7 - X_3_10 X_4_7 - X_4_10 X_5_7 - X_5_10;
      %cons_region(&vars.);

      /* Region 6: */
      %let vars = X_6_1 - X_6_3 X_7_1 - X_7_3
                  X_8_1 - X_8_3 X_9_1 - X_9_3;
      %cons_region(&vars.);

      /* Region 7: */
      %let vars = X_6_4 X_6_5 X_7_4 X_7_5 X_8_4 X_8_5
                  X_9_4 X_9_5 X_10_4 X_10_5 X_11_4 X_11_5;
      %cons_region(&vars.);

      /* Region 8: */
      %let vars = X_6_6 X_6_7 X_7_6 X_7_7 X_8_6 X_8_7
                  X_9_6 X_9_7 X_10_6 X_10_7 X_11_6 X_11_7;
      %cons_region(&vars.);

      /* Region 9: */
      %let vars = X_6_8 X_6_9 X_7_8 X_7_9 X_8_8 X_8_9
                  X_9_8 X_9_9 X_10_8 X_10_9 X_11_8 X_11_9;
      %cons_region(&vars.);

      /* Region 10: */
      %let vars = X_6_10 - X_6_12 X_7_10 - X_7_12
                  X_8_10 - X_8_12 X_9_10 - X_9_12;
      %cons_region(&vars.);

      /* Region 11: */
      %let vars = X_10_1 - X_10_3 X_11_1 - X_11_3 X_12_1 - X_12_6;
      %cons_region(&vars.);

      /* Region 12: */
      %let vars = X_10_10 - X_10_12 X_11_10 - X_11_12 X_12_7 - X_12_12;
      %cons_region(&vars.);
   run;

   %put &_ORCLP_;

%mend pds;

%pds;
```

The only solution of the 2008 Pi Day sudoku puzzle is shown in Output 3.1.4.

Output 3.1.4 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 Output 3.1.5.

Output 3.1.5 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 EVALVARSEL= option to solve a magic square of size seven. When the EVALVARSEL option is specified without a keyword list, the CLP procedure evaluates each of the available variable selection strategies for the amount of time specified by the MAXTIME= option. In this example, MINRMAXC, WDEG, DOMDDEG, and DOMWDEG are the variable selection strategies that find a solution within three seconds. The macro variable _ORCLPEVS_ contains the results for each selection strategy.

```
%macro magic(n);
   %put n = &n;
   /* magic constant */
   %let sum = %eval((&n*(&n*&n+1))/2);
   proc clp out=magic&n evalvarsel maxtime=3;
      /* X_i_j = entry (i,j) */
      %do i = 1 %to &n;
         var (X_&i._1-X_&i._&n) = [1,%eval(&n*&n)];
      %end;
      /* row sums */
      %do i = 1 %to &n;
         lincon 0
         %do j = 1 %to &n;
             + X_&i._&j
         %end;
         = &sum;
      %end;
      /* column sums */
      %do j = 1 %to &n;
         lincon 0
         %do i = 1 %to &n;
            + X_&i._&j
         %end;
         = &sum;
      %end;
      /* diagonal: upper left to lower right */
      lincon 0
      %do i = 1 %to &n;
         + X_&i._&i
      %end;
      = &sum;
      /* diagonal: upper right to lower left */
      lincon 0
      %do i = 1 %to &n;
         + X_%eval(&n+1-&i)_&i
      %end;
      = &sum;
      /* symmetry-breaking */
      lincon X_1_1  + 1 <= X_&n._1;
      lincon X_1_1  + 1 <= X_&n._&n;
      lincon X_1_&n + 1 <= X_&n._1;

      alldiff();
   run;
   %put &_ORCLP_;
   %put &_ORCLPEVS_;
%mend magic;

%magic(7);
```

The solution is displayed in Output 3.1.6.

Output 3.1.6 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 3.2: Alphabet Blocks Problem

This example illustrates usage 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 with an integer variable. The domain of each variable is the set $\{1, 2, 3, 4\}$ that represents block1 through block4. The assignment '$A = 1$' indicates that the letter 'A' is on a side of block1. Each block has six sides; hence each value v in $\{1, 2, 3, 4\}$ has to 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 six.

Moreover, in order to spell all of the words listed previously, the four letters in each of the 12 words have to be on different blocks. Another GCC statement that specifies 12 global cardinality constraints is used to enforce these conditions. You can also formulate these restrictions with 12 alldifferent constraints. Finally, four linear constraints (as specified with LINCON statements) are used to 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 clp out=out;
   /* 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.                                 */
   var (A B C D E F G H I J K L M N O P R S T U V W X Y) = [1,4];
```

```
   /* There are exactly 6 letters on each alphabet block */
   gcc (A B C D E F G H I J K L M N O P R S T U V W X Y) = (
                                                         (1, 6, 6)
                                                         (2, 6, 6)
                                                         (3, 6, 6)
                                                         (4, 6, 6) );

   /* Note 1: Since lv=uv=6 for all v=1,...,4; the above global
              cardinality constraint can also specified as:
   gcc (A B C D E F G H I J K L M N O P R S T U V W X Y) =(DL=6 DU=6);
   */
   /* The letters in each word must be on different blocks. */
   gcc (B A K E) = (DL=0 DU=1)
       (O N Y X) = (DL=0 DU=1)
       (E C H O) = (DL=0 DU=1)
       (O V A L) = (DL=0 DU=1)
       (G I R D) = (DL=0 DU=1)
       (S M U G) = (DL=0 DU=1)
       (J U M P) = (DL=0 DU=1)
       (T O R N) = (DL=0 DU=1)
       (L U C K) = (DL=0 DU=1)
       (V I N Y) = (DL=0 DU=1)
       (L U S H) = (DL=0 DU=1)
       (W R A P) = (DL=0 DU=1);

   /* Note 2: These restrictions can also be enforced by ALLDIFF constraints:
         alldiff (B A K E) (O N Y X) (E C H O) (O V A L)
                 (G I R D) (S M U G) (J U M P) (T O R N)
                 (L U C K) (V I N Y) (L U S H) (W R A P);
   */

   /* 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. */
   lincon B = 1;
   lincon A = 2;
   lincon K = 3;
   lincon E = 4;

run;
```

The solution to this problem is shown in Output 3.2.1.

Output 3.2.1 Solution to Alphabet Blocks Problem

Solution to Alphabet Blocks Problem

Block	Side1	Side2	Side3	Side4	Side5	Side6
1	B	F	I	O	U	W
2	A	C	D	J	N	S
3	H	K	M	R	V	X
4	E	G	L	P	T	Y

Example 3.3: Work-Shift Scheduling Problem

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

Six workers (Alan, Bob, John, Mike, Scott, and Ted) 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 does not work on the first shift; Bob works only on the third shift. The others can work any shift. The objective is to find a feasible assignment for this problem.

You can model the minimum and maximum shift requirements with a GCC constraint and formulate the problem as a standard CSP. The variables W1–W6 identify the shift to be assigned to each of the six workers: Alan, Bob, John, Mike, Scott, and Ted.

```
proc clp out=clpout;
   /*  Six workers (Alan, Bob, John, Mike, Scott and Ted)
       are to be assigned to 3 working shifts.            */
   var (W1-W6) = [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. */
   gcc (W1-W6) = ( ( 1, 1, 4) ( 2, 2, 3) ( 3, 2, 2) );

   /* Alan doesn't work on the first shift. */
   lincon W1 <> 1;

   /* Bob works only on the third shift. */
   lincon W2 = 3;
run;
```

The resulting assignment is shown in Output 3.3.1.

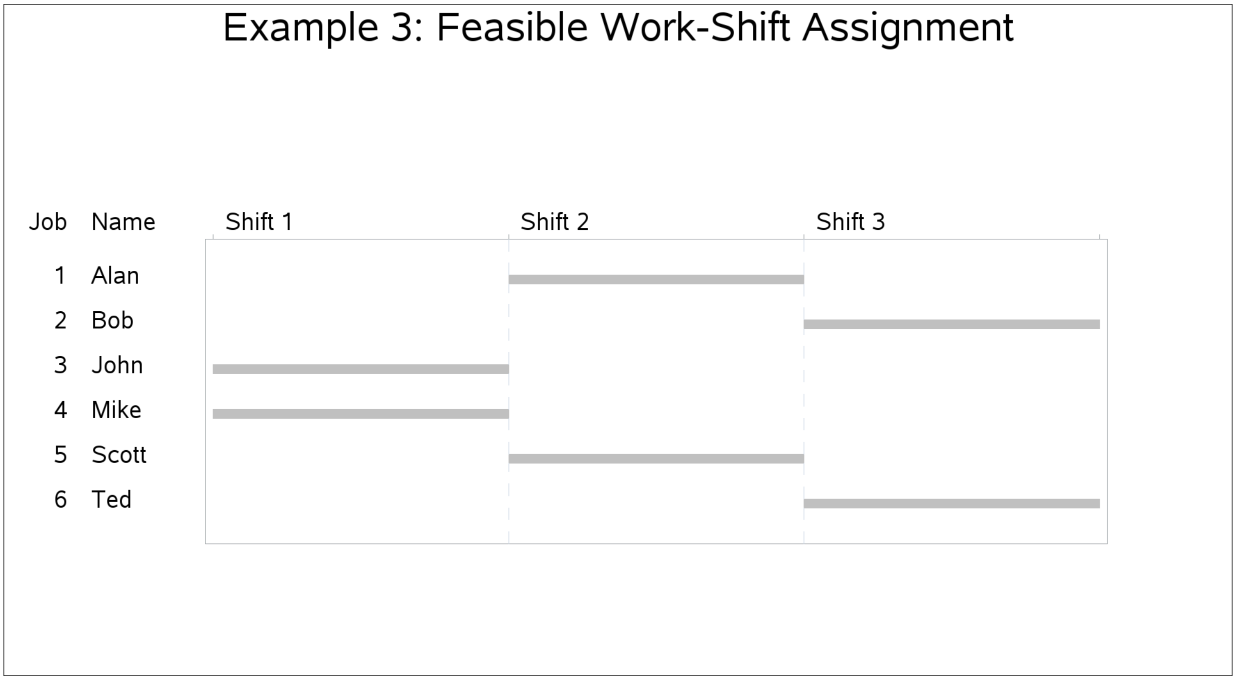
Output 3.3.1 Solution to Work-Shift Scheduling Problem

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 3.3.2.

Output 3.3.2 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 with minimum cost.

The costs of assigning the workers to the different shifts are given in Table 3.9. A dash "-" in position (i, j) indicates that worker i cannot work on shift j.

Table 3.9 Costs of Assigning Workers to Shifts

	Shift 1	**Shift 2**	**Shift 3**
Alan	-	12	10
Bob	-	-	6
John	16	8	12
Mike	10	6	8
Scott	6	6	8
Ted	12	4	4

Based on the cost structure in Table 3.9, the schedule derived previously 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 $\sum_{i=1}^{n} C_i \leq obj$, you can limit the solutions to feasible schedules that cost no more than *obj*.

You can then create a SAS macro %CALLCLP with *obj* as a parameter that can be called iteratively from a search routine to find an optimal solution. The SAS macro %MINCOST(*lb*,*ub*) uses a bisection search to find the minimum cost schedule among all schedules that cost between *lb* and *ub*. Although a value of *ub* = 100 is used in this example, it would suffice to use *ub* = 54, the cost of the feasible schedule determined earlier.

```
%macro callclp(obj);
   %put The objective value is: &obj..;
   proc clp out=clpout;
      /* Six workers (Alan, Bob, John, Mike, Scott and Ted)
         are to be assigned to 3 working shifts.  */
      var (W1-W6) = [1,3];
      var (C1-C6) = [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. */
      gcc (W1-W6) = ( ( 1, 1, 4) ( 2, 2, 3) ( 3, 2, 2) );

      /* Alan doesn't work on the first shift. */
      lincon W1 <> 1;

      /* Bob works only on the third shift. */
      lincon W2 = 3;


      /* Specify the costs of assigning the workers to the shifts.
         Use 100 (a large number) to indicate an assignment
         that is not possible.*/
      element (W1, (100,  12, 10), C1);
      element (W2, (100, 100,  6), C2);
      element (W3, ( 16,   8, 12), C3);
      element (W4, ( 10,   6,  8), C4);
      element (W5, (  6,   6,  8), C5);
      element (W6, ( 12,   4,  4), C6);

      /* The total cost should be no more than the given objective value. */
      lincon C1 + C2 + C3 + C4 + C5 + C6 <= &obj;
   run;

   /* when a solution is found,                      */
   /* &_ORCLP_ contains the string SOLUTIONS_FOUND=1   */
   %if %index(&_ORCLP_,  SOLUTIONS_FOUND=1) %then %let clpreturn=SUCCESSFUL;
%mend;

/* Bisection search method to determine the optimal objective value  */
%macro mincost(lb, ub);
   %do %while (&lb<&ub-1);
      %put Currently lb=&lb, ub=&ub..;
      %let newobj=%eval((&lb+&ub)/2);
      %let clpreturn=NOTFOUND;
      %callclp(&newobj);
```

```
      %if &clpreturn=SUCCESSFUL %then %let ub=&newobj;
      %else %let lb=&newobj;
   %end;

   %callclp(&ub);

   %put Minimum possible objective value within given range is &ub.;
   %put Any value less than &ub makes the problem infeasible. ;

   proc print;
   run;
%mend;

/* Find the minimum objective value between 1 and 100. */
%mincost(lb=1, ub=100);
```

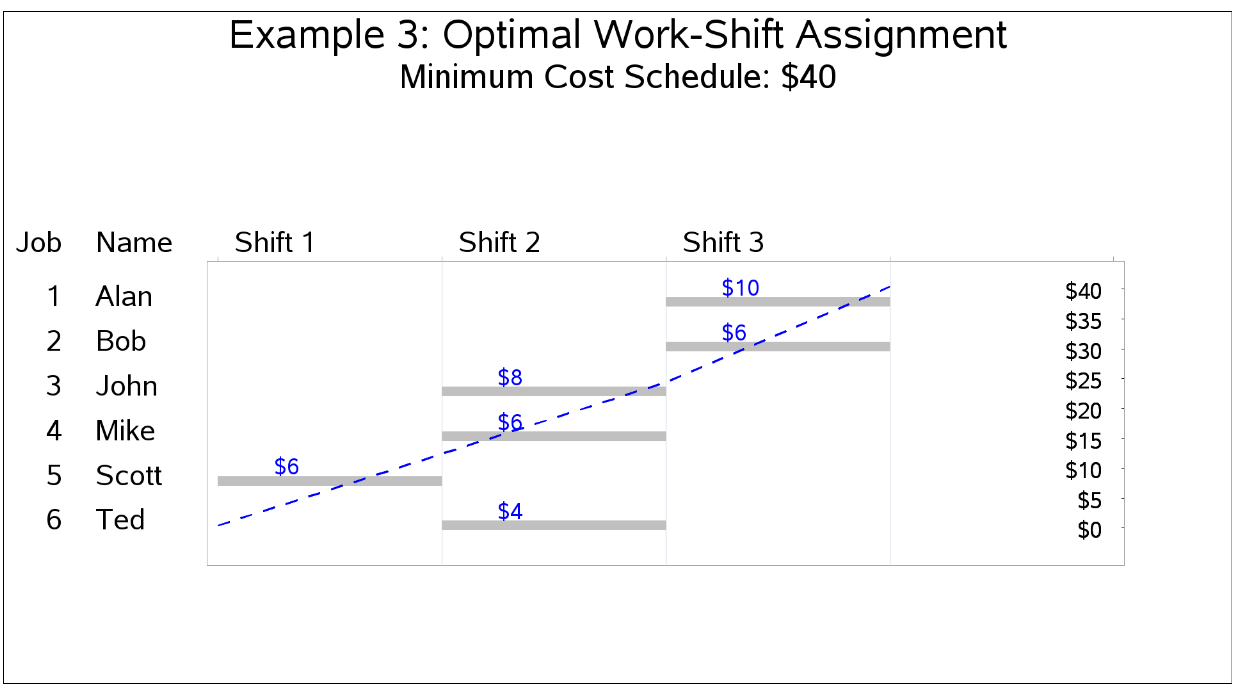
The cost of the optimal schedule, which corresponds to the solution shown in the following output, is 40.

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 3.3.3.

Output 3.3.3 Work-Shift Schedule with Minimum Cost



Example 3.4: 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:

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

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

x_1 : integers in $[-5, 5]$, x_2 : odd integers in $[-5, 9]$, x_3 : integers in $[1, 10]$.

You can use the CLP procedure to 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 $y_4 = \text{mod}(x_1, 4)$. Since the domains of x_1 and x_2 are not consecutive integers that start from 1, you can use element constraints to represent their domains by using index variables z_1 and z_2, respectively. For example, either of the following two ELEMENT constraints specifies that the domain of x_2 is the set of odd integers in $[-5, 9]$:

```
element(z2,(-5,-3,-1,1,3,5,7,9),x2)
element(z2,(-5 to 9 by 2),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. Since 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 $y_1 = x_1^3, x_1 \in [-5, 5]$

```
element(z1,(-125,-64,-27,-8,-1,0,1,8,27,64,125),y1)
```

You can solve the problem in one of the following two ways. The first way is to follow the approach of Example 3.3 by expressing the objective function as a linear constraint $f(x) \ge obj$. Then, you can create a SAS macro %CALLCLP with parameter *obj* and call it iteratively to determine the optimal value of the objective function.

The second way is to define the objective function in the Constraint data set, as demonstrated in the following statements. The data set objdata specifies that the objective function $x_1^3 + 5x_2 - 2^{x_3}$ is to be maximized.

```
data objdata;
   input y1 x2 y2 _TYPE_ $ _RHS_;
/* Objective function: x1^3 + 5 * x2 - 2^x3 */
   datalines;
1 5 -1 MAX .
;

proc clp condata=objdata out=clpout;
   var x1=[-5, 5] x2=[-5, 9] x3=[1, 10] (y1-y4) (z1-z2);

   /* Use element constraint to represent non-contiguous domains */
   /* and nonlinear functions.                                   */
   element
```

```
         /* Domain of x1 is [-5,5] */
         (z1, ( -5 to 5), x1)

         /* Functional Dependencies on x1 */
         /* y1 = x1^3 */
         (z1, (-125, -64, -27, -8, -1, 0, 1, 8, 27, 64, 125), y1)
         /* y4 = mod(x1, 4) */
         (z1, ( -1,  0,  -3,  -2,  -1, 0, 1, 2, 3, 0, 1), y4)

         /* Domain of x2 is the set of odd numbers in [-5, 9] */
         (z2, (-5 to 9 by 2), x2)

         /* Functional Dependencies on x3 */
         /* y2 = 2^x3 */
         (x3, (2, 4, 8, 16, 32, 64, 128, 256, 512, 1024), y2)
         /* y3 = x3^2 */
         (x3, (1, 4, 9, 16, 25, 36, 49, 64, 81, 100), y3);

      lincon
         /* x1 - .5 * x2 + x3^2 <=50 */
         x1 - .5 * x2 + y3 <= 50,

         /* mod(x1, 4) + .25 * x2 >= 1.5 */
         y4 + .25 * x2 >= 1.5;
   run;
   %put &_ORCLP_;
   proc print data=clpout; run;
```

Output 3.4.1 shows the solution that corresponds to the optimal objective value of 168.

Output 3.4.1 Nonlinear Optimization Problem Solution

Obs	x1	x2	x3	y1	y2	y3	y4	z1	z2
1	5	9	1	125	2	1	1	11	8

Example 3.5: 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 variables S_i and C_i represent the ID and color, respectively, of the car in slot i.

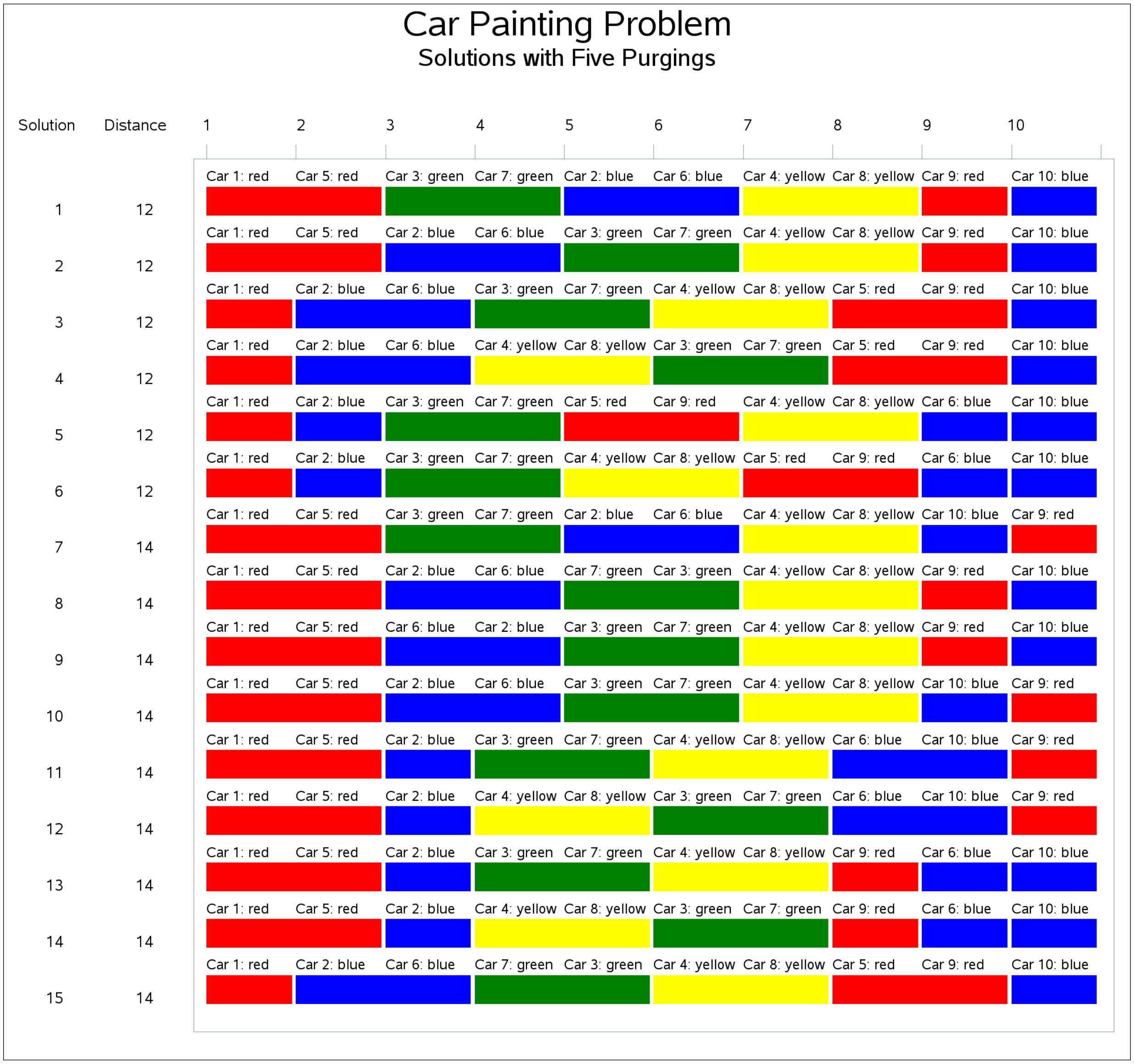
- An element constraint relates the car ID to its color.
- An alldifferent constraint ensures that every slot is associated with a unique car ID.
- Two linear constraints represent the constraint that a car must be painted within three positions of its assembly order.
- The binary variable P_i indicates whether a paint purge takes place after the car in slot i is painted.
- A linear constraint is used to limit the total number of purgings to the required number.

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

```
%macro car_painting(purgings);

    proc clp out=car_ds findall;

       %do i = 1 %to 10;
          var S&i = [1, 10]; /* which car is in slot &i.*/
          var C&i = [1, 4]; /* which color the car in slot &i is.*/
          /* Red=1; Blue=2; Green=3; Yellow=4 */
          element (S&i, (1, 2, 3, 4, 1, 2, 3, 4, 1, 2), C&i);
       %end;

       /* A car can be painted only once. */
       alldiff (S1-S10);

       /* A car must be painted within 3 positions of its assembly order. */
       %do i = 1 %to 10;
          lincon S&i-&i>=-3;
          lincon S&i-&i<=3;
       %end;

       %do i = 1 %to 9;
          var P&i = [0, 1]; /* Whether there is a purge after slot &i*/
          reify P&i: (C&i <> C%eval(&i+1));
       %end;

       /* Calculate the number of purgings. */
       lincon 0
       %do i = 1 %to 9;
          + P&i
       %end;
       <=&purgings ;
    run;

%mend;
%car_painting(5)
```

The problem is infeasible for four purgings. The CLP procedure finds 87 possible solutions for the five-purgings problem. The solutions are sorted by the total distance 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 3.5.1. Each row represents a solution, and each color transition represents a paint purging.

Output 3.5.1 Car Painting Schedule with Five Purgings



Example 3.6: 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 3.6.1. 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 3.6.1 The 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 five 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$j$_$k$ be a binary variable that equals 1 if scene j is shot on day k. Let Ai_k be another binary variable that equals 1 if actor i is present on day k. The variable Name$i$ is the name of the ith actor; Cost$i$ is the daily cost of the ith actor. Ai_Sj = 1 if actor i appears in scene j, and 0 otherwise.

The objective function representing the total production cost is given by

$$\min \sum_{i=1}^{11} \sum_{k=1}^{4} \text{Cost}i \times Ai\_k$$

The %SCENE macro first reads the data set scene and produces three sets of macro variables: Name$i$, Cost$i$, and Ai_Sj. The data set cost is created next to specify the objective function. Finally, the CLP procedure is invoked. There are two sets of GCC constraints in the CLP call: one to make sure each scene is shot exactly once, and one to limit the number of scenes shot per day to be at least four and at most five. There are two sets of LINCON constraints: one to indicate that an actor must be present if any of his or her scenes are shot that day, and one for breaking symmetries to improve efficiency. Additionally, an OBJ statement specifies upper and lower bounds on the objective function to be minimized.

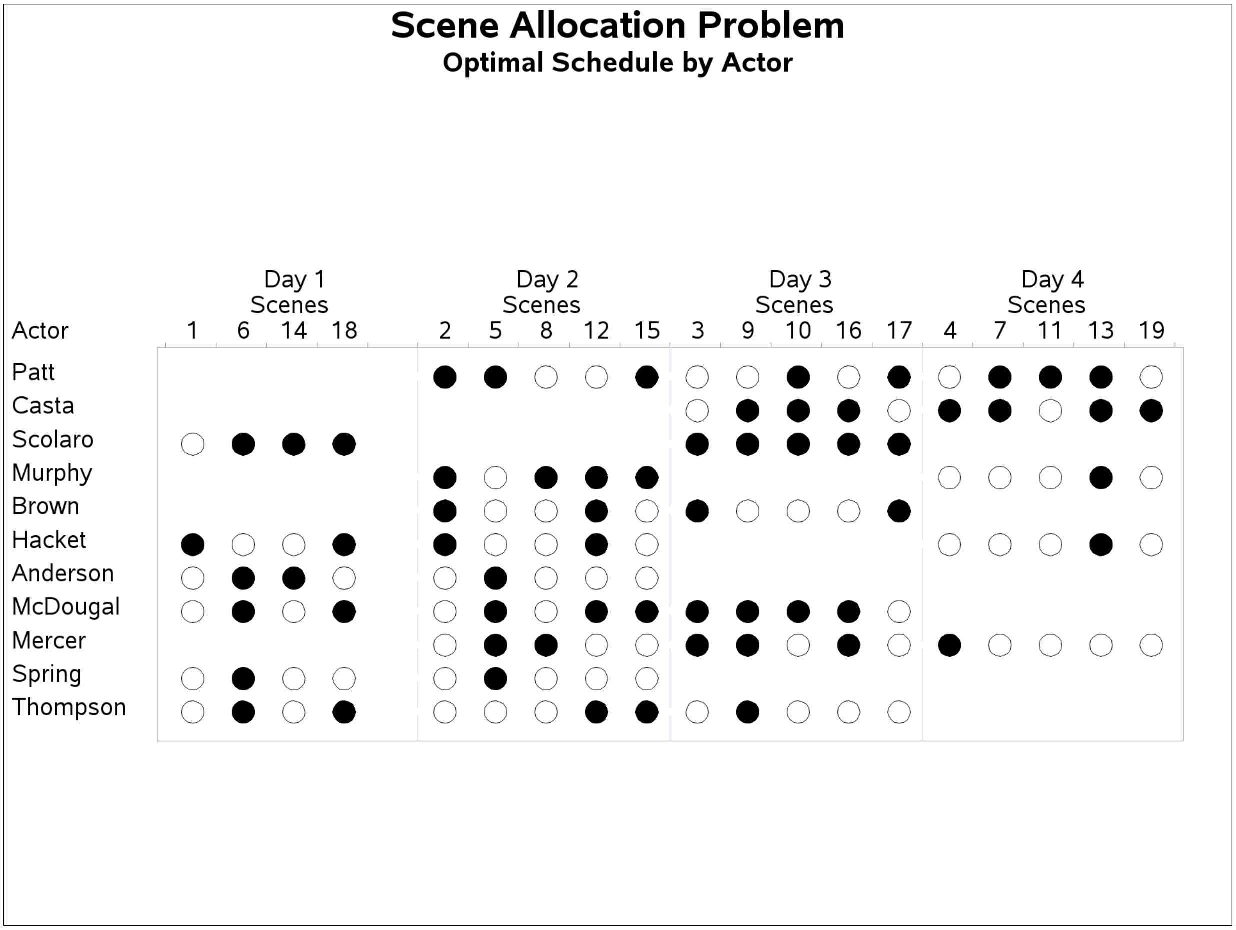
```
%macro scene;
   /* Ai_Sj=1 if actor i appears in scene j    */
   /* Ai_Sj=0 otherwise                        */
   /* Initialize to 0                          */
   %do i=1 %to 11; /* 11 actors */
      %do j=1 %to 19; /* 19 scenes */
         %let A&i._S&j=0;
      %end;
   %end;

   data scene_cost;
      set scene;
      keep DailyFee A;
      retain DailyFee A;
```

```
   do day=1 to 4;
      A='A'||left(strip(_N_))||'_'||left(strip(day));
      output;
   end;
   call symput("Name"||strip(_n_), Actor); /* read actor name */
   call symput("Cost"||strip(_n_), DailyFee); /* read actor cost */
   /* read whether an actor appears in a scene */
   %do i=1 %to 9; /* 9 scene variables in the data set */
      if S_Var&i > 0 then
         call symput("A"||strip(_n_)||"_S"||strip(S_Var&i), 1);
   %end;
run;
/* Create constraint data set which defines the objective function */
proc transpose data=scene_cost out=cost(drop=_name_);
   var DailyFee;
   id A;
run;
data cost;
   set cost;
   _TYPE_='MIN';
   _RHS_=.;
run;

/* Find the minimum objective value.                    */
proc clp condata=cost out=out varselect=maxc;
   /* 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.   */
   obj lb=137739 ub=550956;

   /* Declare variables. */
   %do k=1 %to 4; /* 4 days */
   %do j=1 %to 19; /* 19 scenes */
      var S&j._&k=[0,1]; /* Indicates if scene j is shot on day k.   */
   %end;
   %do i=1 %to 11; /* 11 actors */
      var A&i._&k=[0,1]; /* Indicates if actor i is present on day k.*/
   %end;
   %end;

   /* Every scene is shot exactly once.*/
   %do j=1 %to 19;
      gcc (
         %do k=1 %to 4;
            S&j._&k
         %end;
         )=( (1, 1, 1) );
   %end;

   /* At least 4 and at most 5 scenes are shot per day. */
   %do k=1 %to 4;
      gcc (
         %do j=1 %to 19;
            S&j._&k
```

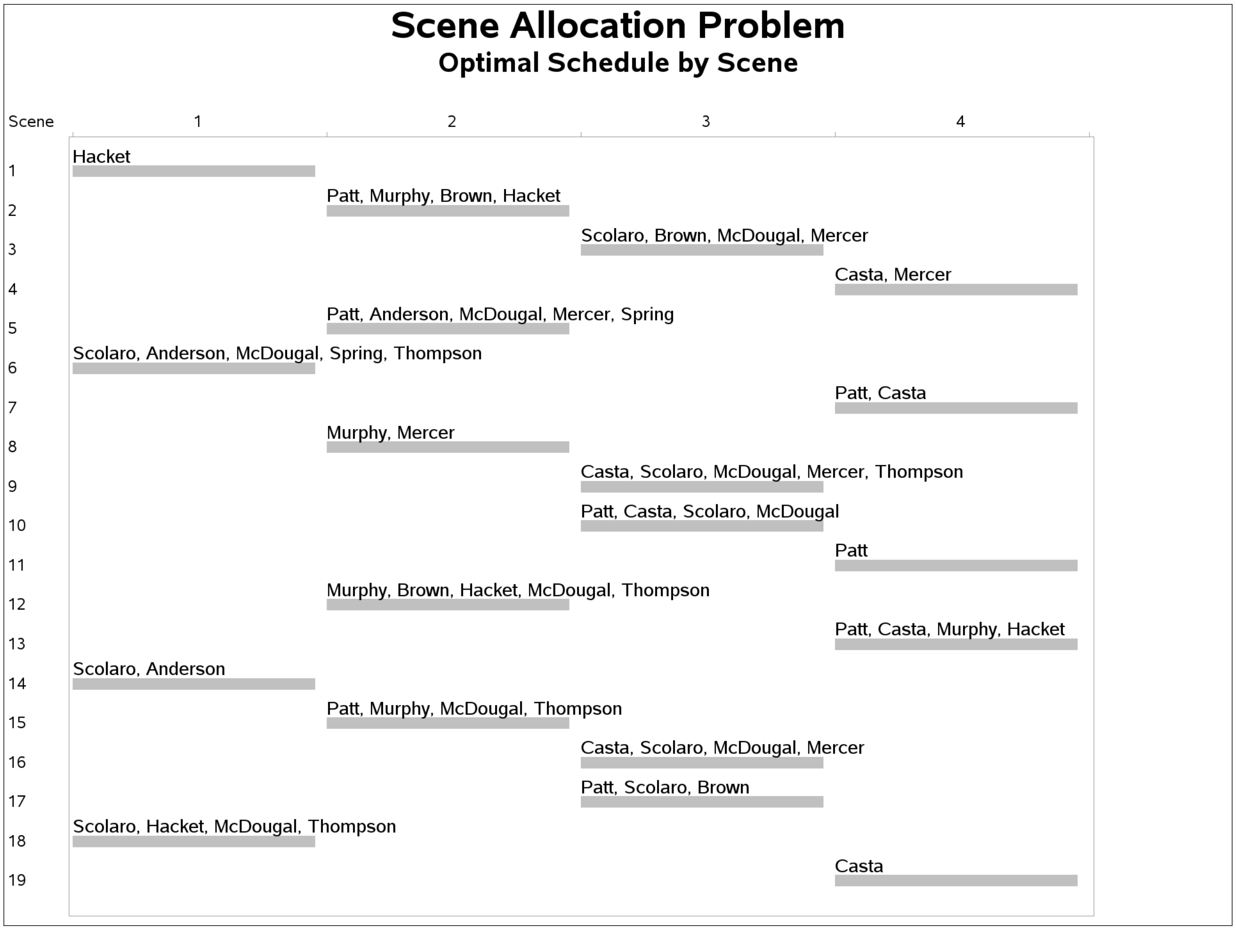
```
            %end;
            )=( (1, 4, 5) );
      %end;

      /* Actors for a scene must be present on day of shooting.*/
      %do k=1 %to 4;
         %do j=1 %to 19;
            %do i=1 %to 11;
               %if (&&A&i._S&j>0) %then %do;
                  lincon S&j._&k <= A&i._&k;
               %end;
            %end;
         %end;
      %end;

      /* Symmetry breaking constraints. Without loss of any generality, we */
      /* 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.  */
      lincon S1_1 = 1, S1_2 = 0, S1_3 = 0, S1_4 = 0,
         S2_3 = 0, S2_4 = 0, S3_4 = 0;

      /* If Scene2 is shot on day 1,                 */
      /* then Scene3 can be shot on day 1 or day 2. */
      lincon S2_1 + S3_3 <= 1;

   run;
   %put &_ORCLP_;

%mend scene;
```

The optimal production cost is 334,144, as reported in the _ORCLP_ macro variable. The corresponding actor schedules and scene schedules are displayed in Output 3.6.2 and Output 3.6.3, respectively.

Output 3.6.2 Scene Allocation Problem: Actor Schedules



Output 3.6.3 Scene Allocation Problem: Scene Schedules



Example 3.7: Car Sequencing Problem

This example is an instance of a category of problems known as the car sequencing problem. There is a considerable amount of literature related to 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 instance data are presented in Table 3.10.

Table 3.10 The Instance Data

Option Name	Option Type	Capacity m/N	Car Class 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 in turn is processed to generate the SAS macro variables shown in Table 3.11 that are used in the CLP procedure. The assembly line is treated as a sequence of slots, and each car must be allocated to a single slot.

Table 3.11 SAS Macro Variables

Macro Variable	Description	Value
Ncars	Number of cars (slots)	10
Nops	Number of options	5
Nclss	Number of classes	6
Max_1–Max_5	For each option, the maximum number of cars with that option in a block	1 2 1 2 1
Blsz_1–Blsz_5	For each option, the block size to which the maximum number refers	2 3 3 5 5
class_1–class_6	Index number of each class	1 2 3 4 5 6
cars_cls_1–cars_cls_6	Number of cars in each class	1 1 2 2 2 2
list_1–list_5	Class indicator list for each option; for example, classes 1, 5, and 6 that require option 1 (*list_1*)	list_1=1,0,0,0,1,1 list_2=0,0,1,1,0,1 list_3=1,0,0,0,1,0 list_4=1,1,0,1,0,0 list_5=0,0,1,0,0,0
cars_opts_1–cars_opts_5	Number of cars for each option (cars_opts_1 represents the number of cars that require option 1)	5 6 3 4 2

The decision variables for this problem are shown in Table 3.12.

Table 3.12 The Decision Variables

Variable Definition	Description
S_1–S_10=[1,6]	S_*i* is the class of cars assigned to slot *i*.
O_1_1–O_1_5=[0,1] . . . O_10_1–O_10_5=[0,1]	O_*i*_*j*=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.

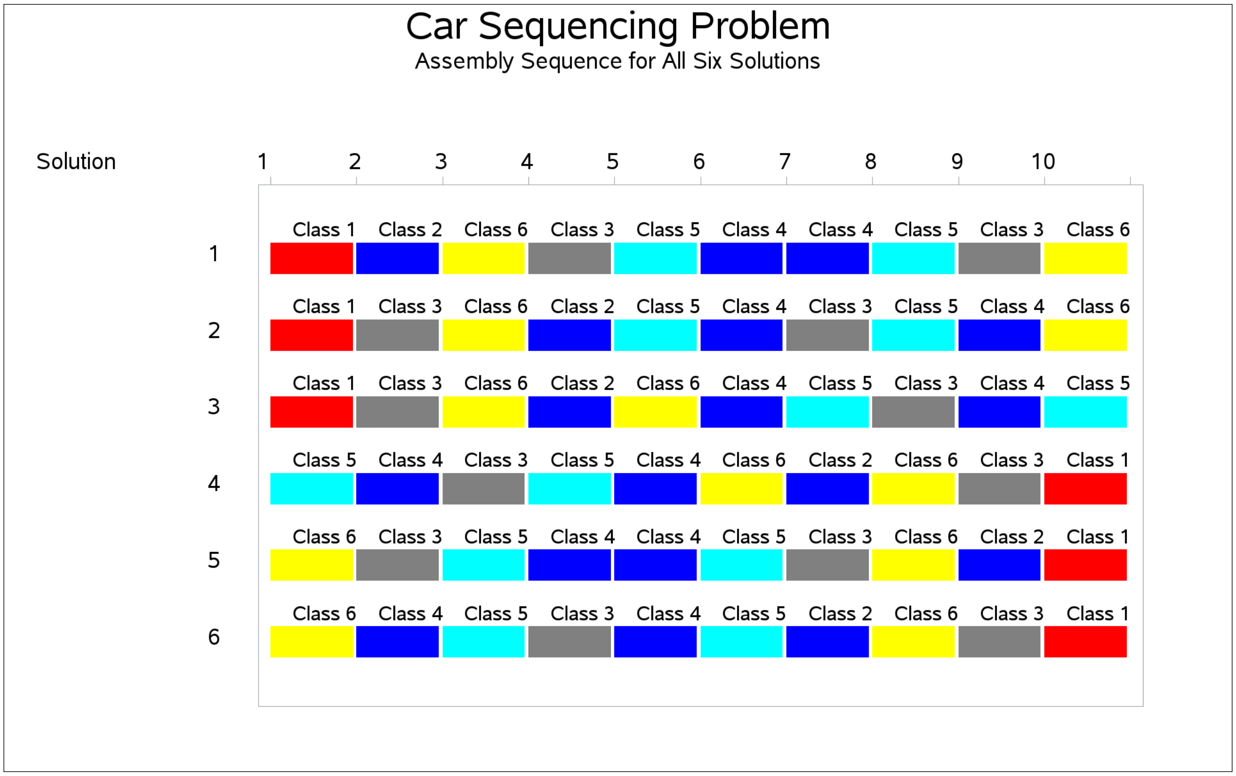
Finally, a set of redundant constraints are introduced to improve the efficiency of propagation. The idea behind the redundant constraint in this model is the following realization: if the workstation for option j has capacity r out of s, then at most r cars in the sequence $(n - s + 1), \ldots, n$ can 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, $k = 1, \ldots, \lfloor n/s \rfloor$.

```
%macro car_sequencing(outdata);

proc clp out=&outdata varselect=minrmaxc findall;

   /* Declare Variables */
   var
      /* Slot variables: Si - class of car assigned to Slot i */
      %do i = 1 %to &Ncars;
          S_&i = [1, &Nclss]
      %end;

      /* Option variables: Oij
         - indicates if class assigned to Sloti needs Option j */
      %do i = 1 %to &Ncars;
          %do j = 1 %to &Nops;
              O_&i._&j = [0, 1]
          %end;
      %end;
      ;

   /* Capacity Constraints: for each option j */
   /* Installed in at most Max_j cars out of every sequence of BlSz_j cars */
   %do j = 1 %to &Nops;
      %do i = 0 %to %eval(&Ncars-&&BlSz_&j);
         lincon 0
         %do k=1 %to &&BlSz_&j;
            + O_%eval(&i+&k)_&j
         %end;
         <=&&Max_&j;
```

```
      %end;
   %end;

   /* Demand Constraints: for each class i */
   /* Exactly cars_cls_i cars */
   gcc (S_1-S_&Ncars) = (
      %do i = 1 %to &Nclss;
         (&&class_&i, &&cars_cls_&i, &&cars_cls_&i)
      %end;
      );

   /* Element Constraints: For each slot i and each option j */
   /* relate the slot variable to the option variables.      */
   /*  O_i_j is the S_i th element in list_j.                */
   %do i = 1 %to &Ncars;
      %do j = 1 %to &Nops;
         element (S_&i, (&&list_&j), O_&i._&j);
      %end;
   %end;


   /* 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                                     */
   %do j = 1 %to &Nops;
      %do i = 1 %to %eval(&Ncars/&&BlSz_&j);
         lincon 0
         %do k=1 %to %eval(&Ncars-&i*&&BlSz_&j);
            + O_&k._&j
         %end;
         >= %eval(&&cars_opts_&j-&i*&&Max_&j);
      %end;
   %end;
run;

%mend;
%car_sequencing(sequence_out);
```

This problem has six solutions, as shown in Output 3.7.1.

Output 3.7.1 Car Sequencing



Example 3.8: Round-Robin Problem

Round-robin tournaments (and variations of them) are a popular format in the scheduling of many sports league tournaments. In a single round-robin tournament, each team plays every other team just once. In a double round-robin (DRR) tournament, each team plays every other team twice: once at home and once away.

This particular example deals with a single round-robin tournament by modeling it as a scheduling CSP. A special case of a double round-robin tournament can be found in Example 3.12, "Scheduling a Major Basketball Conference" and features a different modeling approach.

Consider 14 teams that participate in a single round-robin tournament. Four rooms are provided for the tournament. Thus, $\binom{14}{2} = 91$ games and $\lceil \frac{91}{4} \rceil = 23$ time slots (rounds) need to be scheduled. Since each game requires two teams, a room, and an available time slot, you can regard each game as an activity, the two teams and the room as resources required by the activity, and the time slot as defined by the start and finish times of the activity.

In other words, you can treat this as a scheduling CSP with activities ACT_*i*_*j* where $i < j$, and resources TEAM1 through TEAM14 and ROOM1 through ROOM4. For a given *i* and *j*, activity ACT_*i*_*j* requires the resources TEAM*i*, TEAM*j*, and one of ROOM1 through ROOM4. The resulting start time for activity A_*i*_*j* is the time slot for the game between TEAM*i* and TEAM*j* and the assigned ROOM is the venue for the game.

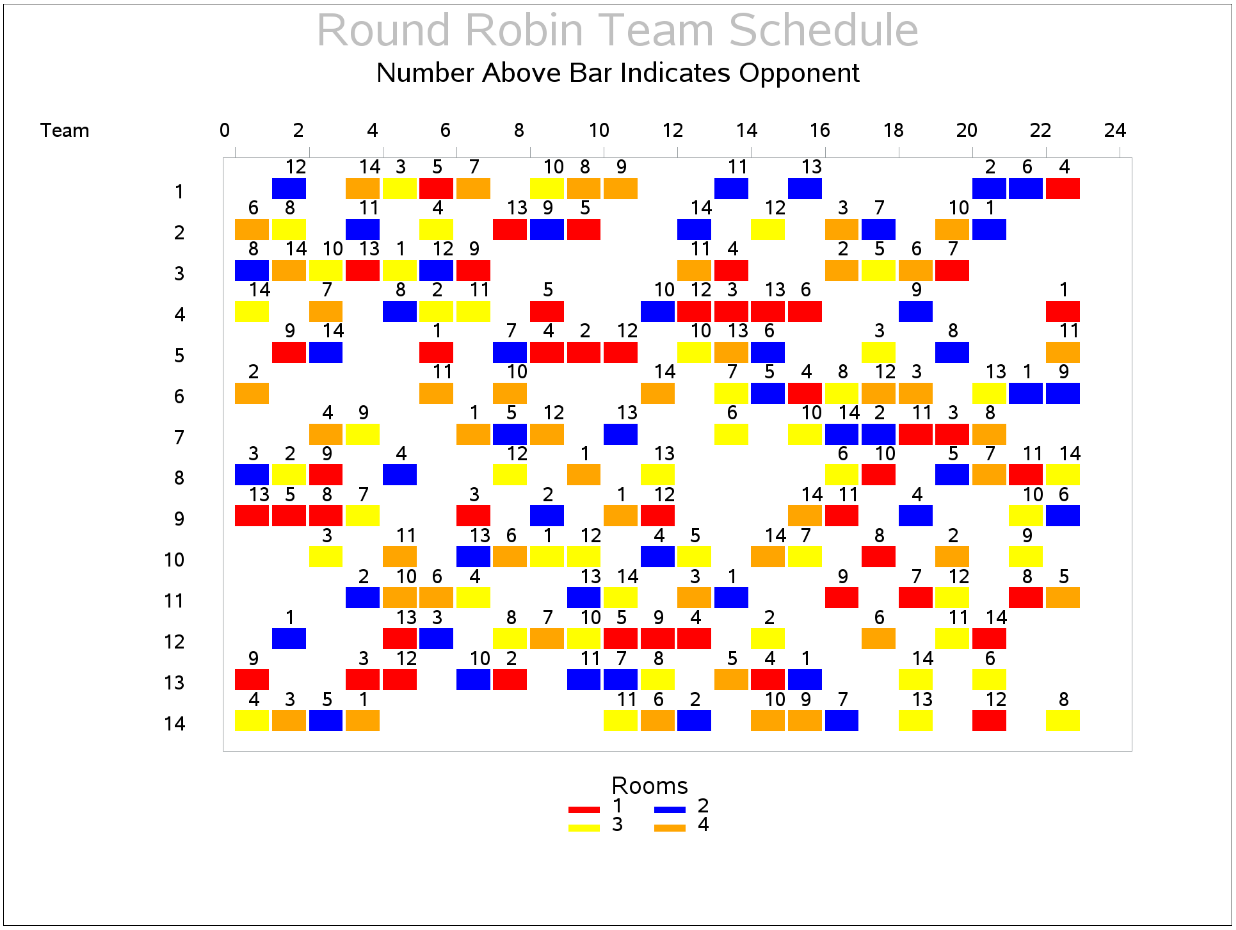
The following SAS macro, %ROUND_ROBIN, uses the CLP procedure to solve this problem. The %ROUND_ROBIN macro uses the number of teams as a parameter.

The ACTDATA= data set defines all activities ACT_*i*_*j* with duration one. The RESOURCE statement declares the TEAM and ROOM resources. The REQUIRES statement defines the resource requirements for each activity ACT_*i*_*j*. The SCHEDULE statement defines the activity selection strategy as MINLS, which selects an activity with minimum late start time from the set of activities that begin prior to the earliest early finish time.

```
%macro round_robin(nteams);

   %let nrounds = %eval(%sysfunc(ceil((&nteams * (&nteams - 1)/2)/4)));

   data actdata;
      do i = 1 to &nteams - 1;
         do j = i + 1 to &nteams;
            _activity_ = compress('ACT_'||put(i,best.)||'_'||put(j,best.));
            _duration_ = 1;
            output;
         end;
      end;
   run;

   proc clp actdata = actdata schedule = schedule;
      schedule finish = &nrounds actselect=minls;
      resource (TEAM1-TEAM&nteams);
      resource (ROOM1-ROOM4);
      requires
         %do i = 1 %to &nteams - 1;
            %do j = &i + 1 %to &nteams;
               ACT_&i._&j = ( TEAM&i )
               ACT_&i._&j = ( TEAM&j )
               ACT_&i._&j = ( ROOM1, ROOM2, ROOM3, ROOM4)
            %end;
         %end;
      ;
      run;

   proc sort data=schedule;
      by start finish;
   run;

%mend round_robin;

%round_robin(14);
```

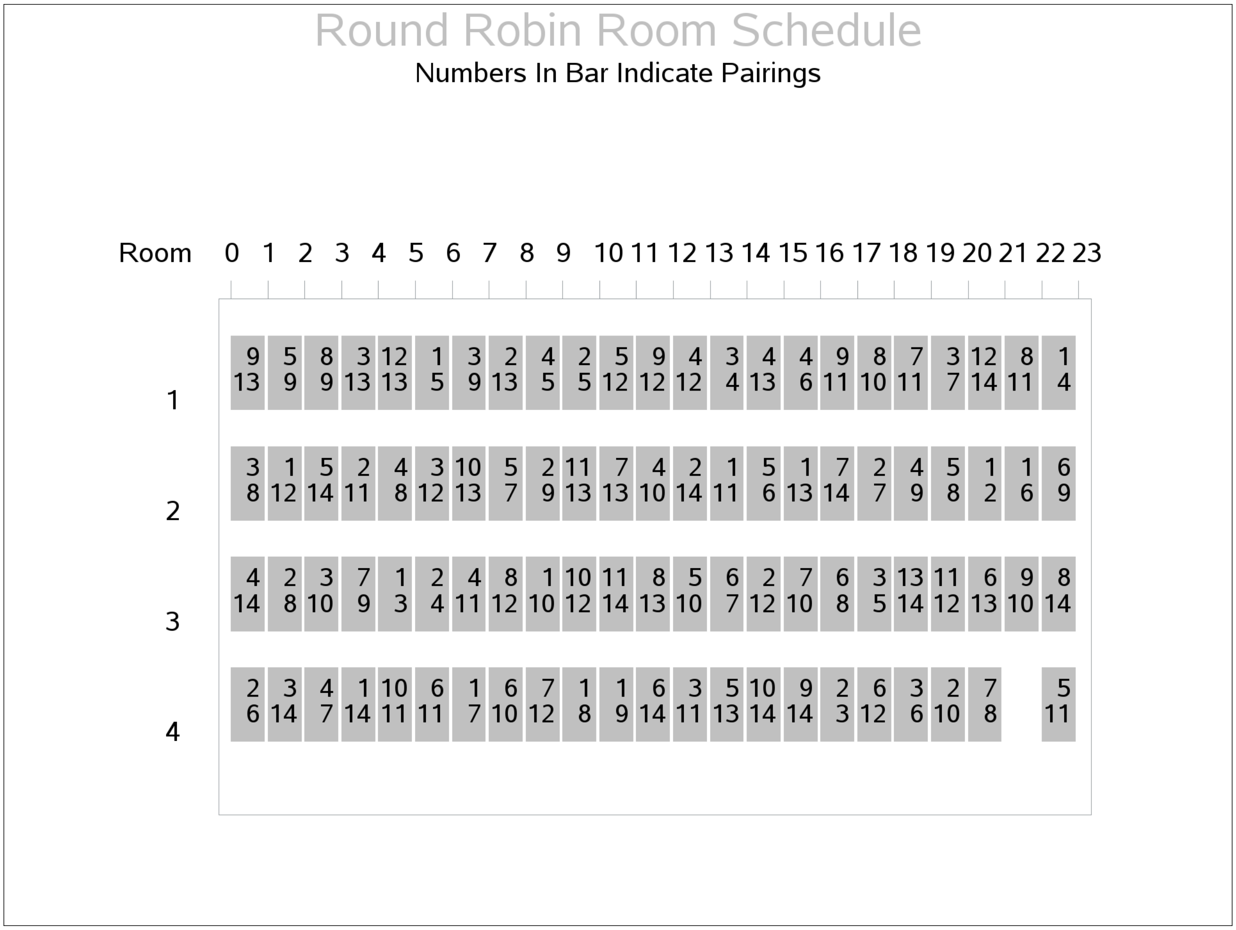
The resulting team schedule is displayed in Output 3.8.1. The vertical axis lists the teams, and the horizontal axis indicates the time slot of each game. The color of the bar indicates the room the game is played in, while the text above each bar identifies the opponent.

Output 3.8.1 Round Robin Team Schedule



Another view of the complete schedule is the room schedule, which is shown in Output 3.8.2. The vertical axis lists each room, and the horizontal axis indicates the time slot of each game. The numbers inside each bar identify the team pairings for the corresponding room and time slot.

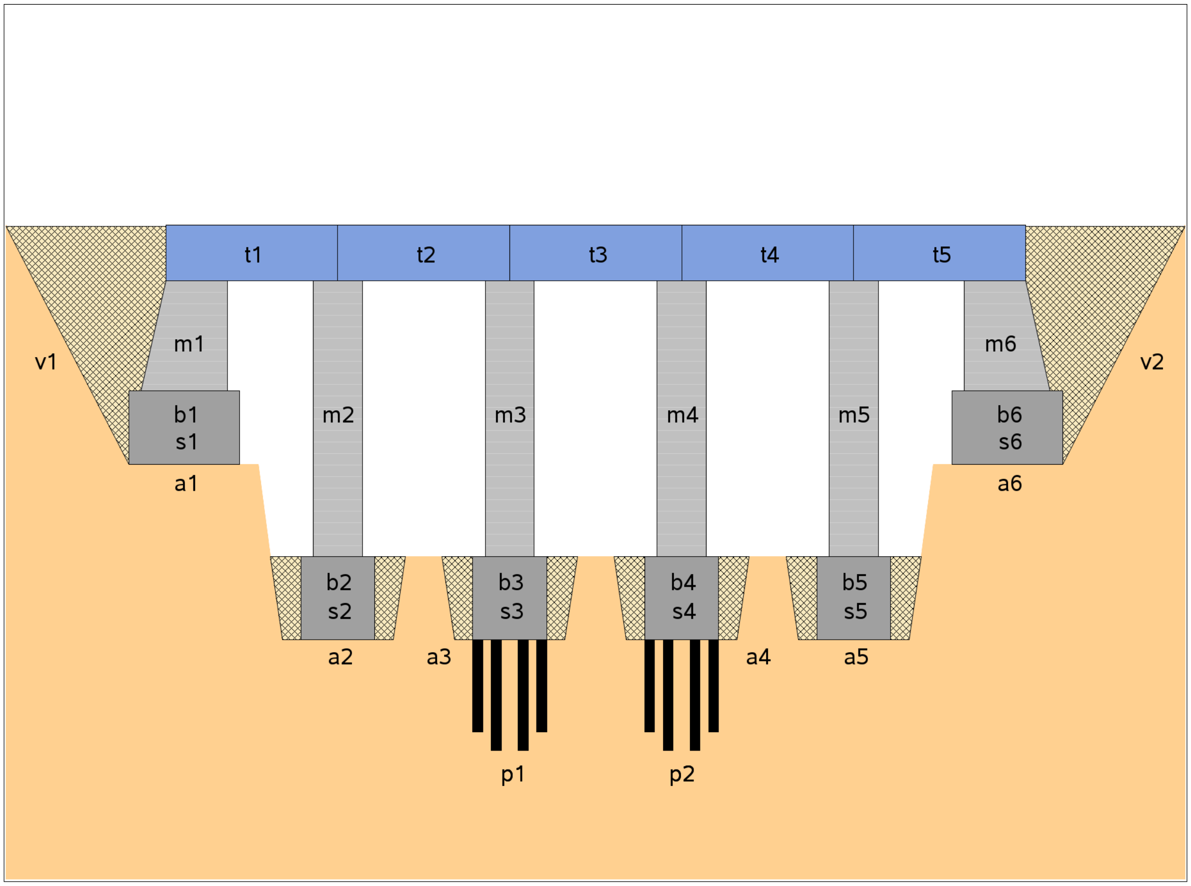
Output 3.8.2 Round Robin Room Schedule



Example 3.9: Resource-Constrained Scheduling with Nonstandard Temporal Constraints

This example illustrates a real-life scheduling problem and is used as a benchmark problem in the constraint programming community. The problem is to schedule the construction of a five-segment bridge. (See Output 3.9.1.) It comes from a Ph.D. dissertation on scheduling problems (Bartusch 1983).

Output 3.9.1 The Bridge Problem



The project consists of 44 tasks and a set of constraints that relate these tasks. Table 3.13 displays the activity information, standard precedence constraints, and resource constraints. The sharing of a unary resource by multiple activities results in the resource constraints being disjunctive in nature.

Table 3.13 Data for Bridge Construction

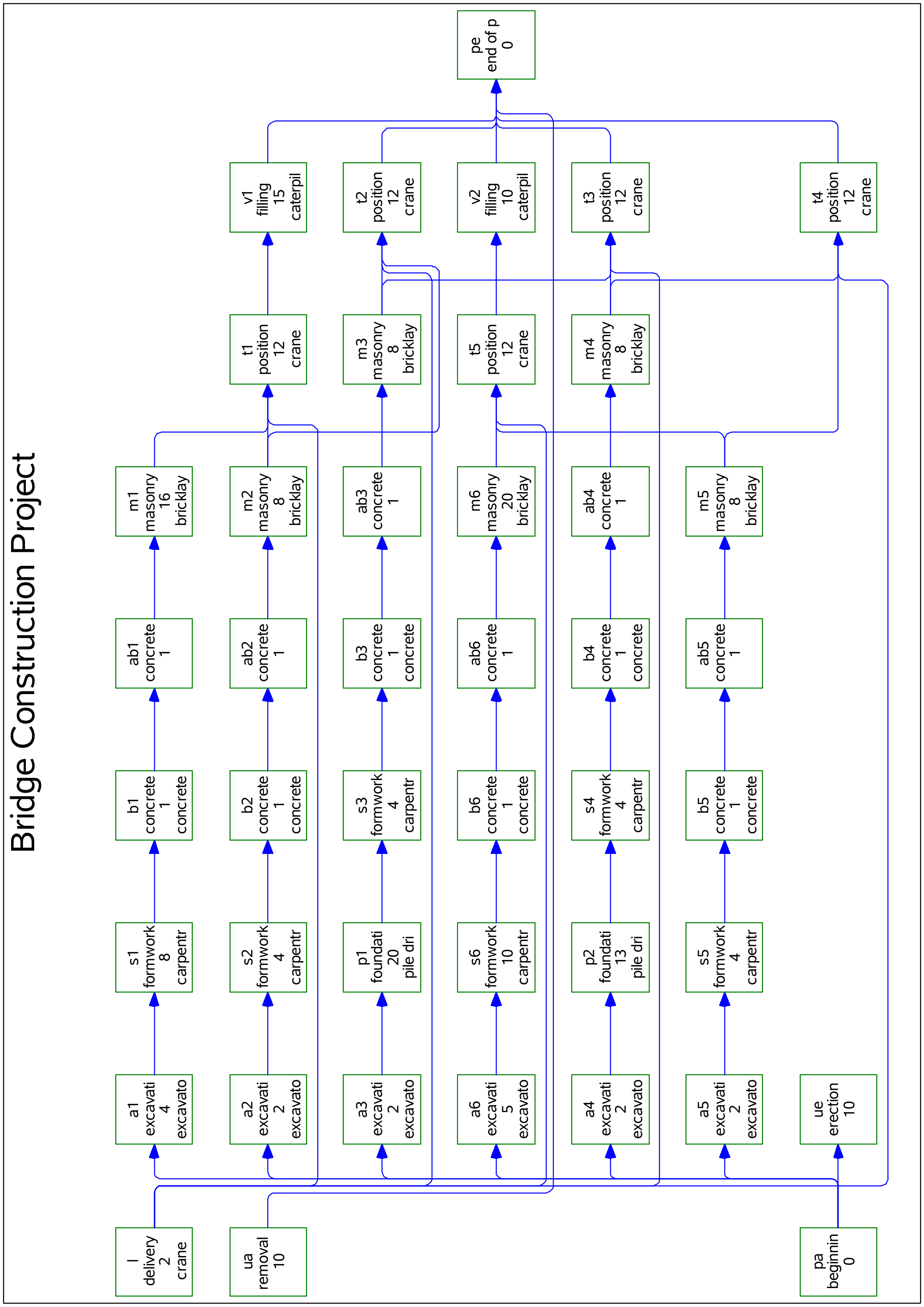
Activity	Description	Duration	Predecessors	Resource
pa	Beginning of project	0		
a1	Excavation (abutment 1)	4	pa	Excavator
a2	Excavation (pillar 1)	2	pa	Excavator
a3	Excavation (pillar 2)	2	pa	Excavator
a4	Excavation (pillar 3)	2	pa	Excavator
a5	Excavation (pillar 4)	2	pa	Excavator
a6	Excavation (abutment 2)	5	pa	Excavator
p1	Foundation piles 2	20	a3	Pile driver
p2	Foundation piles 3	13	a4	Pile driver
ue	Erection of temporary housing	10	pa	
s1	Formwork (abutment 1)	8	a1	Carpentry
s2	Formwork (pillar 1)	4	a2	Carpentry

Table 3.13 *continued*

Activity	Description	Duration	Predecessors	Resource
s3	Formwork (pillar 2)	4	p1	Carpentry
s4	Formwork (pillar 3)	4	p2	Carpentry
s5	Formwork (pillar 4)	4	a5	Carpentry
s6	Formwork (abutment 2)	10	a6	Carpentry
b1	Concrete foundation (abutment 1)	1	s1	Concrete mixer
b2	Concrete foundation (pillar 1)	1	s2	Concrete mixer
b3	Concrete foundation (pillar 2)	1	s3	Concrete mixer
b4	Concrete foundation (pillar 3)	1	s4	Concrete mixer
b5	Concrete foundation (pillar 4)	1	s5	Concrete mixer
b6	Concrete foundation (abutment 2)	1	s6	Concrete mixer
ab1	Concrete setting time (abutment 1)	1	b1	
ab2	Concrete setting time (pillar 1)	1	b2	
ab3	Concrete setting time (pillar 2)	1	b3	
ab4	Concrete setting time (pillar 3)	1	b4	
ab5	Concrete setting time (pillar 4)	1	b5	
ab6	Concrete setting time (abutment 2)	1	b6	
m1	Masonry work (abutment 1)	16	ab1	Bricklaying
m2	Masonry work (pillar 1)	8	ab2	Bricklaying
m3	Masonry work (pillar 2)	8	ab3	Bricklaying
m4	Masonry work (pillar 3)	8	ab4	Bricklaying
m5	Masonry work (pillar 4)	8	ab5	Bricklaying
m6	Masonry work (abutment 2)	20	ab6	Bricklaying
l	Delivery of the preformed bearers	2		Crane
t1	Positioning (preformed bearer 1)	12	m1, m2, l	Crane
t2	Positioning (preformed bearer 2)	12	m2, m3, l	Crane
t3	Positioning (preformed bearer 3)	12	m3, m4, l	Crane
t4	Positioning (preformed bearer 4)	12	m4, m5, l	Crane
t5	Positioning (preformed bearer 5)	12	m5, m6, l	Crane
ua	Removal of the temporary housing	10		
v1	Filling 1	15	t1	Caterpillar
v2	Filling 2	10	t5	Caterpillar
pe	End of project	0	t2, t3, t4, v1, v2, ua	

Output 3.9.2 shows a network diagram that illustrates the precedence constraints in this problem. Each node represents an activity and gives the activity code, truncated description, duration, and the required resource, if any. The network diagram is generated using the SAS/OR NETDRAW procedure.

Output 3.9.2 Network Diagram for the Bridge Construction Project



The following constraints are in addition to the standard precedence constraints:

1. The time between the completion of a particular formwork and the completion of its corresponding concrete foundation is at most four days:

$$f(\text{s}i) \geq f(\text{b}i) - 4, \quad i = 1, \ldots, 6$$

2. There are at most three days between the end of a particular excavation (or foundation piles) and the beginning of the corresponding formwork:

$$f(\text{a}i) \geq s(\text{s}i) - 3, \quad i = 1, 2, 5, 6$$

and

$$f(\text{p1}) \geq s(\text{s3}) - 3$$

$$f(\text{p2}) \geq s(\text{s4}) - 3$$

3. The formworks must start at least six days after the beginning of the erection of the temporary housing:

$$s(\text{s}i) \geq s(\text{ue}) + 6, \quad i = 1, \ldots, 6$$

4. The removal of the temporary housing can start at most two days before the end of the last masonry work:

$$s(\text{ua}) \geq f(\text{m}i) - 2, \quad i = 1, \ldots, 6$$

5. The delivery of the preformed bearers occurs exactly 30 days after the beginning of the project:

$$s(\text{l}) = s(\text{pa}) + 30$$

The following DATA step defines the data set bridge, which encapsulates all of the precedence constraints and also indicates the resources that are required by each activity. Note the use of the reserved variables _ACTIVITY_, _SUCCESSOR_, _LAG_, and _LAGDUR_ to define the activity and precedence relationships. The list of reserved variables can be found in Table 3.6. The latter two variables are required for the nonstandard precedence constraints listed previously.

```
data bridge;
   format _ACTIVITY_ $3. _DESC_ $34. _RESOURCE_ $14.
          _SUCCESSOR_ $3. _LAG_ $3. ;
   input _ACTIVITY_ & _DESC_ & _DURATION_ _RESOURCE_ &
          _SUCCESSOR_ & _LAG_ & _LAGDUR_;
   _QTY_ = 1;
   datalines;
a1   excavation (abutment 1)            4  excavator       s1   .     .
a2   excavation (pillar 1)              2  excavator       s2   .     .
a3   excavation (pillar 2)              2  excavator       p1   .     .
a4   excavation (pillar 3)              2  excavator       p2   .     .
a5   excavation (pillar 4)              2  excavator       s5   .     .
a6   excavation (abutment 2)            5  excavator       s6   .     .
ab1  concrete setting time (abutment 1)  1  .               m1   .     .
```

```
ab2  concrete setting time (pillar 1)     1  .               m2   .     .
ab3  concrete setting time (pillar 2)     1  .               m3   .     .
ab4  concrete setting time (pillar 3)     1  .               m4   .     .
ab5  concrete setting time (pillar 4)     1  .               m5   .     .
ab6  concrete setting time (abutment 2)   1  .               m6   .     .
b1   concrete foundation (abutment 1)     1  concrete mixer  ab1  .     .
b1   concrete foundation (abutment 1)     1  concrete mixer  s1   ff   -4
b2   concrete foundation (pillar 1)       1  concrete mixer  ab2  .     .
b2   concrete foundation (pillar 1)       1  concrete mixer  s2   ff   -4
b3   concrete foundation (pillar 2)       1  concrete mixer  ab3  .     .
b3   concrete foundation (pillar 2)       1  concrete mixer  s3   ff   -4
b4   concrete foundation (pillar 3)       1  concrete mixer  ab4  .     .
b4   concrete foundation (pillar 3)       1  concrete mixer  s4   ff   -4
b5   concrete foundation (pillar 4)       1  concrete mixer  ab5  .     .
b5   concrete foundation (pillar 4)       1  concrete mixer  s5   ff   -4
b6   concrete foundation (abutment 2)     1  concrete mixer  ab6  .     .
b6   concrete foundation (abutment 2)     1  concrete mixer  s6   ff   -4
l    delivery of the preformed bearers    2  crane           t1   .     .
l    delivery of the preformed bearers    2  crane           t2   .     .
l    delivery of the preformed bearers    2  crane           t3   .     .
l    delivery of the preformed bearers    2  crane           t4   .     .
l    delivery of the preformed bearers    2  crane           t5   .     .
m1   masonry work (abutment 1)           16  bricklaying     t1   .     .
m1   masonry work (abutment 1)           16  bricklaying     ua   fs   -2
m2   masonry work (pillar 1)              8  bricklaying     t1   .     .
m2   masonry work (pillar 1)              8  bricklaying     t2   .     .
m2   masonry work (pillar 1)              8  bricklaying     ua   fs   -2
m3   masonry work (pillar 2)              8  bricklaying     t2   .     .
m3   masonry work (pillar 2)              8  bricklaying     t3   .     .
m3   masonry work (pillar 2)              8  bricklaying     ua   fs   -2
m4   masonry work (pillar 3)              8  bricklaying     t3   .     .
m4   masonry work (pillar 3)              8  bricklaying     t4   .     .
m4   masonry work (pillar 3)              8  bricklaying     ua   fs   -2
m5   masonry work (pillar 4)              8  bricklaying     t4   .     .
m5   masonry work (pillar 4)              8  bricklaying     t5   .     .
m5   masonry work (pillar 4)              8  bricklaying     ua   fs   -2
m6   masonry work (abutment 2)           20  bricklaying     t5   .     .
m6   masonry work (abutment 2)           20  bricklaying     ua   fs   -2
p1   foundation piles 2                  20  pile driver     s3   .     .
p2   foundation piles 3                  13  pile driver     s4   .     .
pa   beginning of project                 0  .               a1   .     .
pa   beginning of project                 0  .               a2   .     .
pa   beginning of project                 0  .               a3   .     .
pa   beginning of project                 0  .               a4   .     .
pa   beginning of project                 0  .               a5   .     .
pa   beginning of project                 0  .               a6   .     .
pa   beginning of project                 0  .               l    fse  30
pa   beginning of project                 0  .               ue   .     .
pe   end of project                       0  .               .    .     .
s1   formwork (abutment 1)                8  carpentry       b1   .     .
s1   formwork (abutment 1)                8  carpentry       a1   sf   -3
s2   formwork (pillar 1)                  4  carpentry       b2   .     .
s2   formwork (pillar 1)                  4  carpentry       a2   sf   -3
s3   formwork (pillar 2)                  4  carpentry       b3   .     .
```

```
s3   formwork (pillar 2)               4  carpentry     p1   sf   -3
s4   formwork (pillar 3)               4  carpentry     b4   .     .
s4   formwork (pillar 3)               4  carpentry     p2   sf   -3
s5   formwork (pillar 4)               4  carpentry     b5   .     .
s5   formwork (pillar 4)               4  carpentry     a5   sf   -3
s6   formwork (abutment 2)            10  carpentry     b6   .     .
s6   formwork (abutment 2)            10  carpentry     a6   sf   -3
t1   positioning (preformed bearer 1) 12  crane         v1   .     .
t2   positioning (preformed bearer 2) 12  crane         pe   .     .
t3   positioning (preformed bearer 3) 12  crane         pe   .     .
t4   positioning (preformed bearer 4) 12  crane         pe   .     .
t5   positioning (preformed bearer 5) 12  crane         v2   .     .
ua   removal of the temporary housing 10  .             pe   .     .
ue   erection of temporary housing    10  .             .    .     .
ue   erection of temporary housing    10  .             s1   ss    6
ue   erection of temporary housing    10  .             s2   ss    6
ue   erection of temporary housing    10  .             s3   ss    6
ue   erection of temporary housing    10  .             s4   ss    6
ue   erection of temporary housing    10  .             s5   ss    6
ue   erection of temporary housing    10  .             s6   ss    6
v1   filling 1                        15  caterpillar   pe   .     .
v2   filling 2                        10  caterpillar   pe   .     .
;
```

The CLP procedure is then invoked by using the following statements with the SCHEDTIME= option.

```
/* invoke PROC CLP */
proc clp actdata=bridge schedtime=schedtime_bridge;
   schedule finish=104;
run;
```

The FINISH= option is specified to find a solution in 104 days, which also happens to be the optimal makespan.

The schedtime_bridge data set contains the activity start and finish times as computed by the CLP procedure. Since an activity gets assigned to at most one resource, it is possible to represent the complete schedule information more concisely by merging the schedtime_bridge data set with the bridge_info data set, as shown in the following statements.

```
/* Create Consolidated Schedule */
proc sql;
   create table bridge_info as
          select distinct _ACTIVITY_ as ACTIVITY format $3. length 3,
          _DESC_ as DESCRIPTION, _RESOURCE_ as RESOURCE from bridge;

proc sort data=schedtime_bridge;
   by ACTIVITY;
run;

data schedtime_bridge;
   merge bridge_info schedtime_bridge;
   by ACTIVITY;
run;
```

```
proc sort data=schedtime_bridge;
   by START FINISH;
run;

proc print data=schedtime_bridge noobs width=min;;
   title 'Bridge Construction Schedule';
run;
```

Output 3.9.3 shows the resulting merged data set.

Output 3.9.3 Bridge Construction Schedule

Bridge Construction Schedule

ACTIVITY	DESCRIPTION	RESOURCE	SOLUTION	DURATION	START	FINISH
pa	beginning of project		1	0	0	0
a4	excavation (pillar 3)	excavator	1	2	0	2
ue	erection of temporary housing		1	10	0	10
a5	excavation (pillar 4)	excavator	1	2	2	4
p2	foundation piles 3	pile driver	1	13	2	15
a2	excavation (pillar 1)	excavator	1	2	5	7
s5	formwork (pillar 4)	carpentry	1	4	6	10
a3	excavation (pillar 2)	excavator	1	2	7	9
b5	concrete foundation (pillar 4)	concrete mixer	1	1	10	11
s2	formwork (pillar 1)	carpentry	1	4	10	14
ab5	concrete setting time (pillar 4)		1	1	11	12
a1	excavation (abutment 1)	excavator	1	4	12	16
m5	masonry work (pillar 4)	bricklaying	1	8	12	20
b2	concrete foundation (pillar 1)	concrete mixer	1	1	14	15
ab2	concrete setting time (pillar 1)		1	1	15	16
s4	formwork (pillar 3)	carpentry	1	4	15	19
p1	foundation piles 2	pile driver	1	20	15	35
b4	concrete foundation (pillar 3)	concrete mixer	1	1	19	20
a6	excavation (abutment 2)	excavator	1	5	19	24
s1	formwork (abutment 1)	carpentry	1	8	19	27
ab4	concrete setting time (pillar 3)		1	1	20	21
m2	masonry work (pillar 1)	bricklaying	1	8	20	28
b1	concrete foundation (abutment 1)	concrete mixer	1	1	27	28
s6	formwork (abutment 2)	carpentry	1	10	27	37
ab1	concrete setting time (abutment 1)		1	1	28	29
m4	masonry work (pillar 3)	bricklaying	1	8	28	36
l	delivery of the preformed bearers	crane	1	2	30	32
t4	positioning (preformed bearer 4)	crane	1	12	36	48
m1	masonry work (abutment 1)	bricklaying	1	16	36	52
b6	concrete foundation (abutment 2)	concrete mixer	1	1	37	38
s3	formwork (pillar 2)	carpentry	1	4	37	41
ab6	concrete setting time (abutment 2)		1	1	38	39
b3	concrete foundation (pillar 2)	concrete mixer	1	1	41	42
ab3	concrete setting time (pillar 2)		1	1	42	43
m3	masonry work (pillar 2)	bricklaying	1	8	52	60
t1	positioning (preformed bearer 1)	crane	1	12	52	64
m6	masonry work (abutment 2)	bricklaying	1	20	60	80
t2	positioning (preformed bearer 2)	crane	1	12	64	76
v1	filling 1	caterpillar	1	15	64	79
ua	removal of the temporary housing		1	10	78	88
t5	positioning (preformed bearer 5)	crane	1	12	80	92
v2	filling 2	caterpillar	1	10	92	102
t3	positioning (preformed bearer 3)	crane	1	12	92	104
pe	end of project		1	0	104	104

A Gantt chart of the schedule in Output 3.9.3, produced using the SAS/OR GANTT procedure, is displayed in Output 3.9.4. Each activity bar is color coded according to the resource associated with it. The legend identifies the name of the resource that is associated with each color.

Example 3.10: Scheduling with Alternate Resources

This example shows an interesting job shop scheduling problem that illustrates the use of alternative resources. There are 90 jobs (J1–J90), each taking either one or two days, that need to be processed on one of ten machines (M0–M9). Not every machine can process every job. In addition, certain jobs also require one of seven operators (OP0–OP6). As with the machines, not every operator can be assigned to every job. There are no explicit precedence relationships in this example.

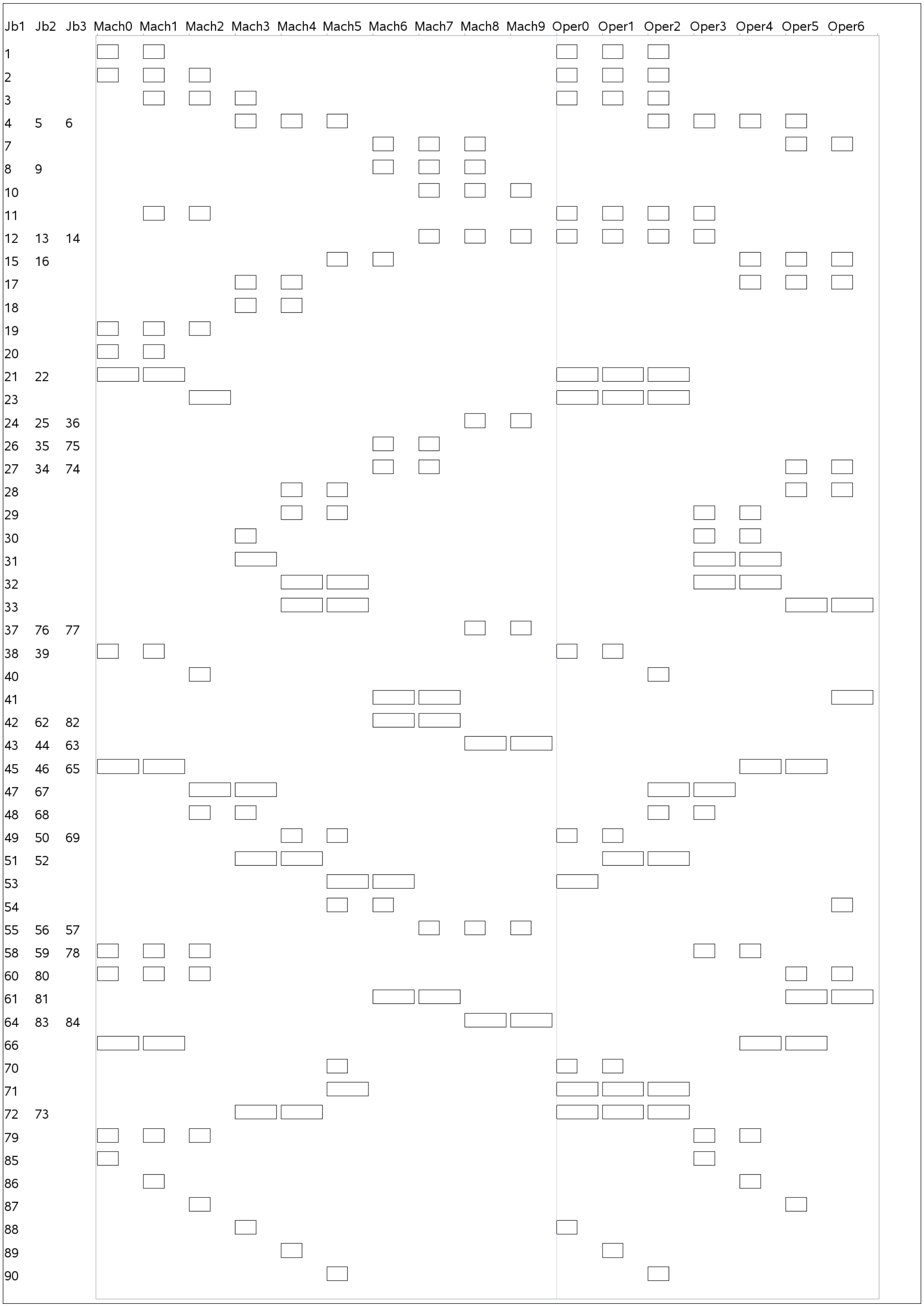
The machine and operator requirements for each job are shown in Output 3.10.1. Each row in the graph defines a resource requirement for up to three jobs that are identified in the columns Jb1–Jb3 to the left of the chart. The horizontal axis of the chart represents the resources and is split into two regions by a vertical line. The resources to the left of the divider are the machines, Mach0–Mach9, and the resources to the right of the divider are the operators, Oper0–Oper6. For each row on the chart, a bar on the chart represents a potential requirement for the corresponding resource listed above.

Each of the jobs listed in columns Jb1–Jb3 can be processed on one of the machines in Mach0–Mach9 and requires the assistance of one of the operators in Oper0–Oper6 while being processed. An eligible resource is represented by a bar, and the length of the bar indicates the duration of the job.

For example, row five specifies that job number 7 can be processed on machine 6, 7, or 8 and additionally requires either operator 5 or operator 6 in order to be processed. The next row indicates that jobs 8 and 9 can also be processed on the same set of machines. However, they do not require any operator assistance.

Output 3.9.4 Gantt Chart for the Bridge Construction Project

Output 3.10.1 Machine and Operator Requirements



The CLP procedure is invoked by using the following statements with FINISH=12 in the SCHEDULE statement to obtain a 12-day solution that is also known to be optimal. In order to obtain the optimal solution, it is necessary to invoke the edge-finding consistency routines, which are activated with the EDGEFINDER option in the SCHEDULE statement. The activity selection strategy is specified as DMINLS, which selects the activity with the earliest late start time. Activities with identical resource requirements are grouped together in the REQUIRES statement.

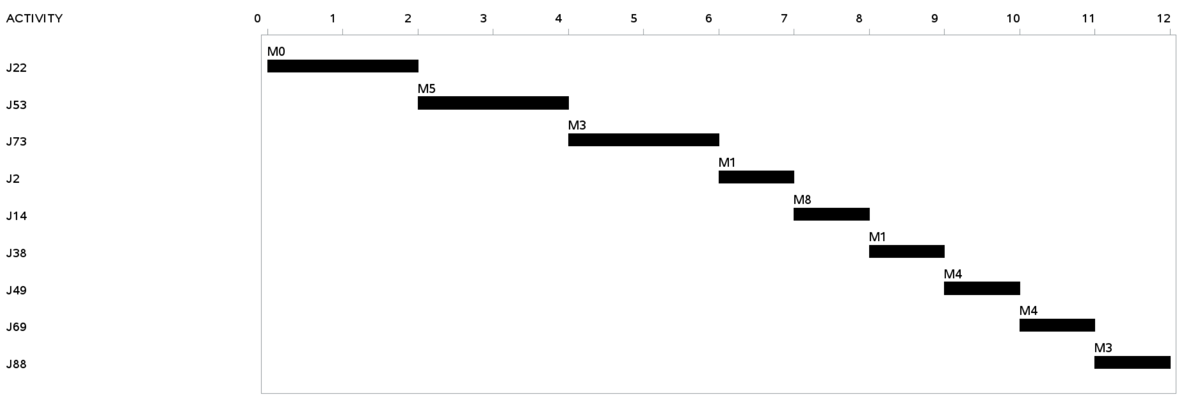
```
proc clp dom=[0,12] restarts=500 dpr=6 showprogress
   schedtime=schedtime_altres schedres=schedres_altres;
   schedule start=0 finish=12 actselect=dminls edgefinder;

   activity (J1-J20 J24-J30 J34-J40 J48-J50 J54-J60
             J68-J70 J74-J80 J85-J90) = (1)    /* one day jobs */
            (J21-J23 J31-J33 J41-J47 J51-J53 J61-J67
             J71-J73 J81-J84) = (2);           /* two day jobs */

   resource (M0-M9) (OP0-OP6);

   requires
      /* machine requirements */
      (J85) = (M0)
      (J1 J20 J21 J22 J38 J39 J45 J46 J65 J66) = (M0, M1)
      (J19 J2 J58 J59 J60 J78 J79 J80) = (M0, M1, M2)
      (J86) = (M1)
      (J11) = (M1, M2)
      (J3) = (M1, M2, M3)
      (J23 J40 J87) = (M2)
      (J47 J48 J67 J68) = (M2, M3)
      (J30 J31 J88) = (M3)
      (J17 J18 J51 J52 J72 J73) = (M3, M4)
      (J4 J5 J6) = (M3, M4, M5)
      (J89) = (M4)
      (J28 J29 J32 J33 J49 J50 J69) = (M4, M5)
      (J70 J71 J90) = (M5)
      (J15 J16 J53 J54) = (M5, M6)
      (J26 J27 J34 J35 J41 J42 J61 J62 J74 J75 J81 J82) = (M6, M7)
      (J7 J8 J9) = (M6, M7, M8)
      (J10 J12 J13 J14 J55 J56 J57) = (M7, M8, M9)
      (J24 J25 J36 J37 J43 J44 J63 J64 J76 J77 J83 J84) = (M8, M9)
      /* operator requirements */
      (J53 J88) = (OP0)
      (J38 J39 J49 J50 J69 J70) = (OP0, OP1)
      (J1 J2 J21 J22 J23 J3 J71 J72 J73) = (OP0, OP1, OP2)
      (J11 J12 J13 J14) = (OP0, OP1, OP2, OP3)
      (J89) = (OP1)
      (J51 J52) = (OP1, OP2)
      (J40 J90) = (OP2)
      (J47 J48 J67 J68) = (OP2, OP3)
      (J4 J5 J6) = (OP2, OP3, OP4, OP5)
      (J85) = (OP3)
      (J29 J30 J31 J32 J58 J59 J78 J79) = (OP3, OP4)
      (J86) = (OP4)
```

```
        (J45 J46 J65 J66) = (OP4, OP5)
        (J15 J16 J17) = (OP4, OP5, OP6)
        (J87) = (OP5)
        (J27 J28 J33 J34 J60 J61 J7 J74 J80 J81) = (OP5, OP6)
        (J41 J54) = (OP6);
run;
```

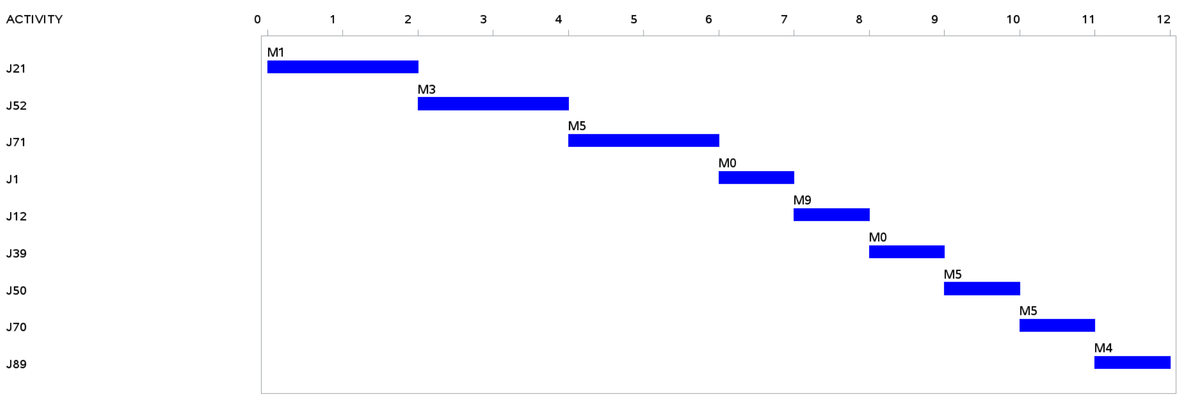
The resulting schedule is shown in a series of Gantt charts that are displayed in Output 3.10.2 and Output 3.10.3. In each of these Gantt charts, the vertical axis lists the different jobs, the horizontal bar represents the start and finish times for each of the jobs, and the text above each bar identifies the machine that the job is being processed on. Output 3.10.2 displays the schedule for the operator-assisted tasks (one for each operator), while Output 3.10.3 shows the schedule for automated tasks (that is, those tasks that do not require operator intervention).

Output 3.10.2 Operator-Assisted Jobs Schedule

Schedule for Operator OP0
Machine Identified Above Bar



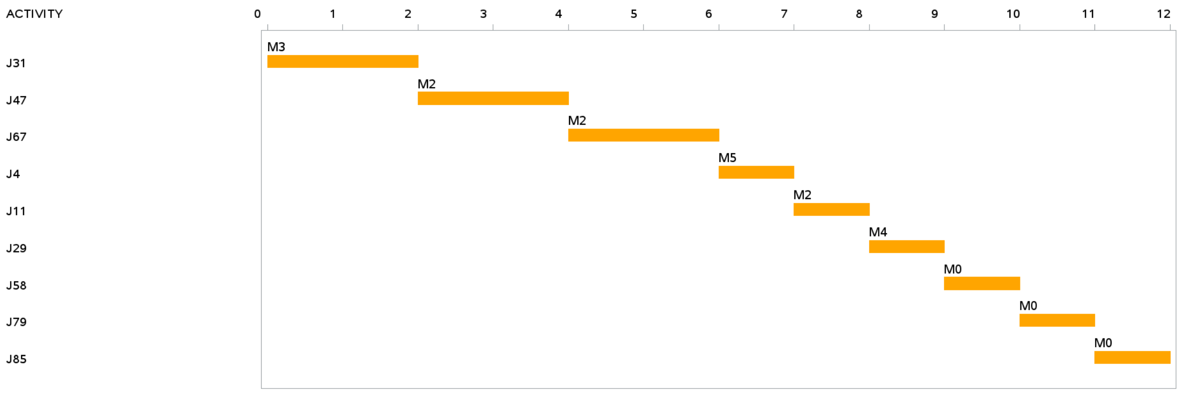
Schedule for Operator OP1
Machine Identified Above Bar



Output 3.10.2 *continued*

Schedule for Operator OP2
Machine Identified Above Bar

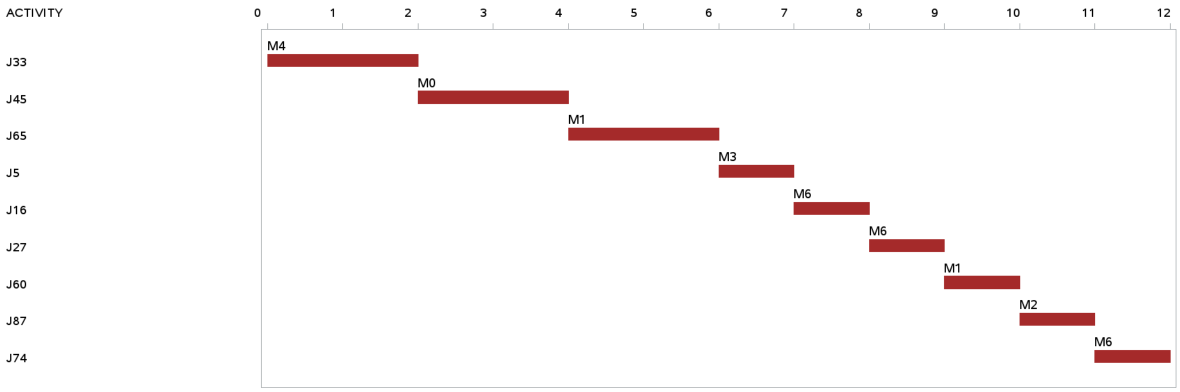
Schedule for Operator OP3
Machine Identified Above Bar



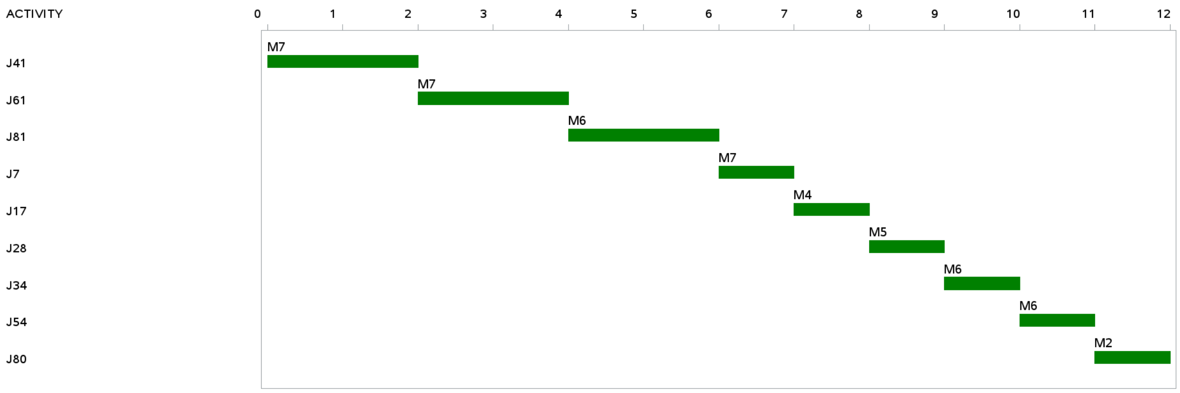
Schedule for Operator OP4
Machine Identified Above Bar

Output 3.10.2 *continued*

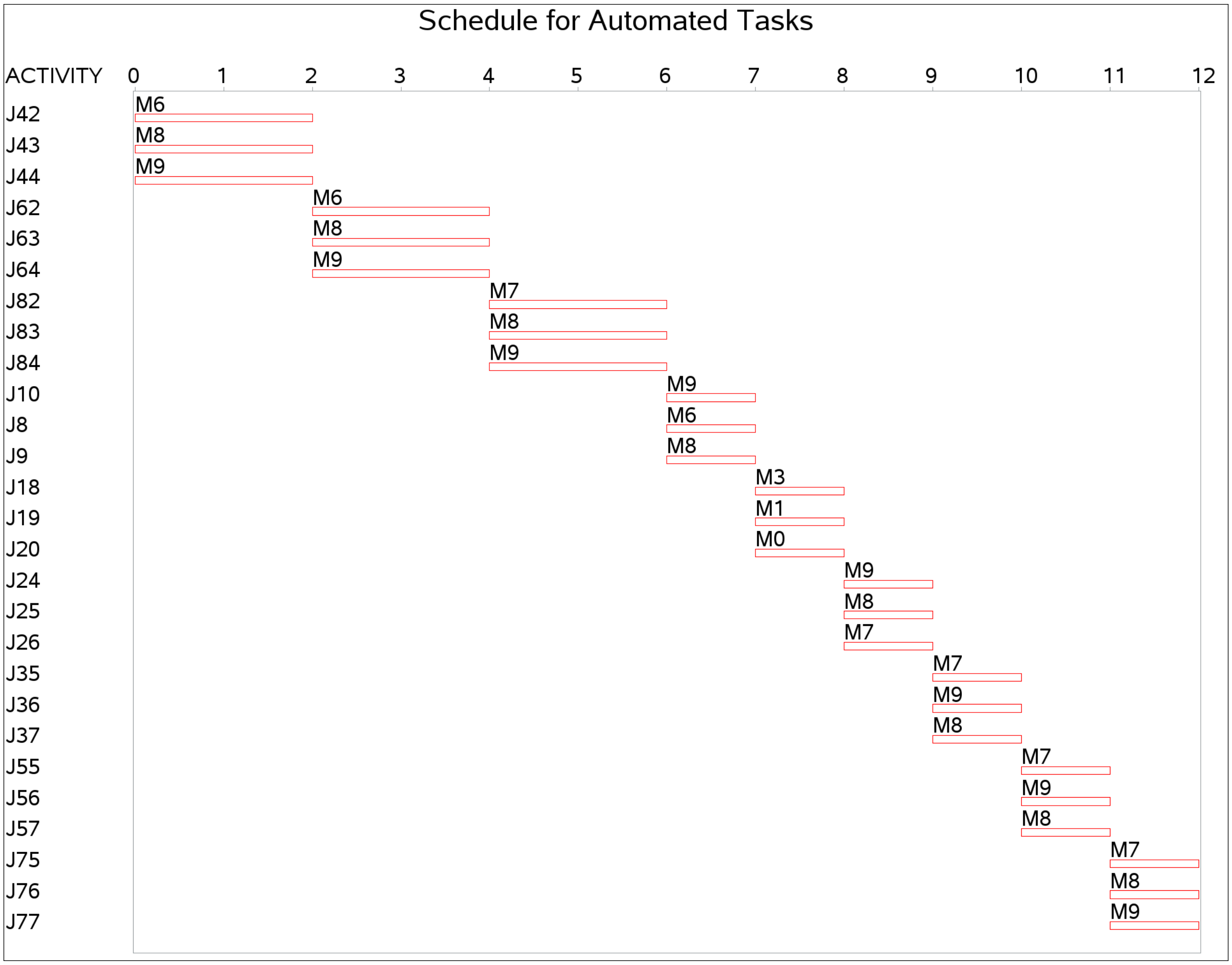
Schedule for Operator OP5
Machine Identified Above Bar



Schedule for Operator OP6
Machine Identified Above Bar

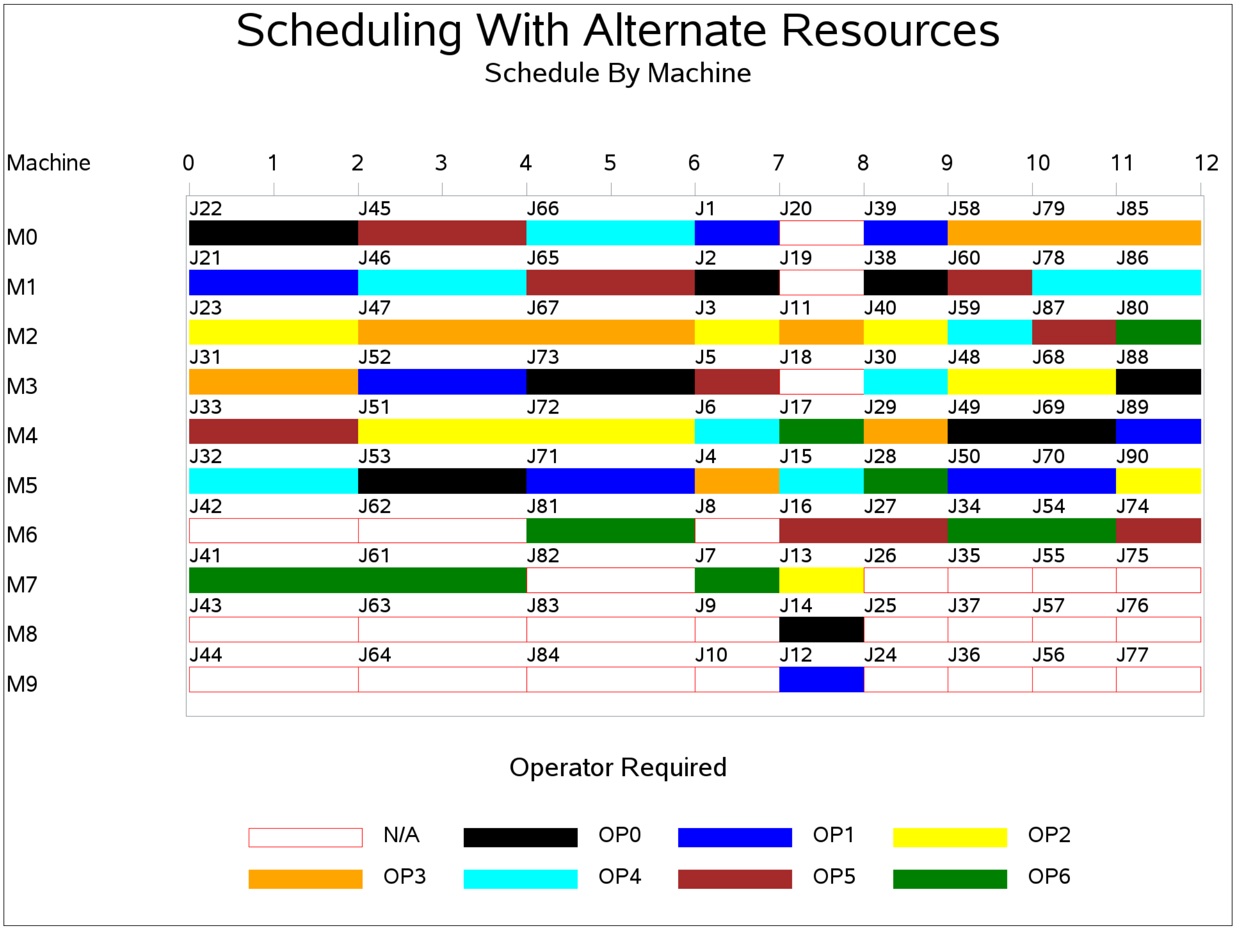


Output 3.10.3 Automated Jobs Schedule



A more interesting Gantt chart is that of the resource schedule by machine, as shown in Output 3.10.4. This chart displays the schedule for each machine. Every row corresponds to a machine. Every bar on each row consists of multiple segments, and every segment represents a job that is processed on the machine. Each segment is also coded according to the operator assigned to it. The mapping of the coding is indicated in the legend. It is evident that the schedule is optimal since none of the machines or operators are idle at any time during the schedule.

Output 3.10.4 Machine Schedule



Example 3.11: 10×10 Job Shop Scheduling Problem

This example is a job shop scheduling problem from Lawrence (1984). This test is also known as LA19 in the literature, and its optimal makespan is known to be 842 (Applegate and Cook 1991). There are 10 jobs (J1–J10) and 10 machines (M0–M9). Every job must be processed on each of the 10 machines in a predefined sequence. The objective is to minimize the completion time of the last job to be processed, known as the makespan. The jobs are described in the data set raw by using the following statements.

```
/* jobs specification */
data raw (drop=i mid);
   do i=1 to 10;
      input mid _DURATION_ @;
      _RESOURCE_=compress('M'||put(mid,best.));
      output;
   end;
   datalines;
2  44  3   5  5  58  4  97  0   9  7  84  8  77  9  96  1  58  6  89
4  15  7  31  1  87  8  57  0  77  3  85  2  81  5  39  9  73  6  21
9  82  6  22  4  10  3  70  1  49  0  40  8  34  2  48  7  80  5  71
1  91  2  17  7  62  5  75  8  47  4  11  3   7  6  72  9  35  0  55
```

```
6  71  1  90  3  75  0  64  2  94  8  15  4  12  7  67  9  20  5  50
7  70  5  93  8  77  2  29  4  58  6  93  3  68  1  57  9   7  0  52
6  87  1  63  4  26  5   6  2  82  3  27  7  56  8  48  9  36  0  95
0  36  5  15  8  41  9  78  3  76  6  84  4  30  7  76  2  36  1   8
5  88  2  81  3  13  6  82  4  54  7  13  8  29  9  40  1  78  0  75
9  88  4  54  6  64  7  32  0  52  2   6  8  54  5  82  3   6  1  26
;
```

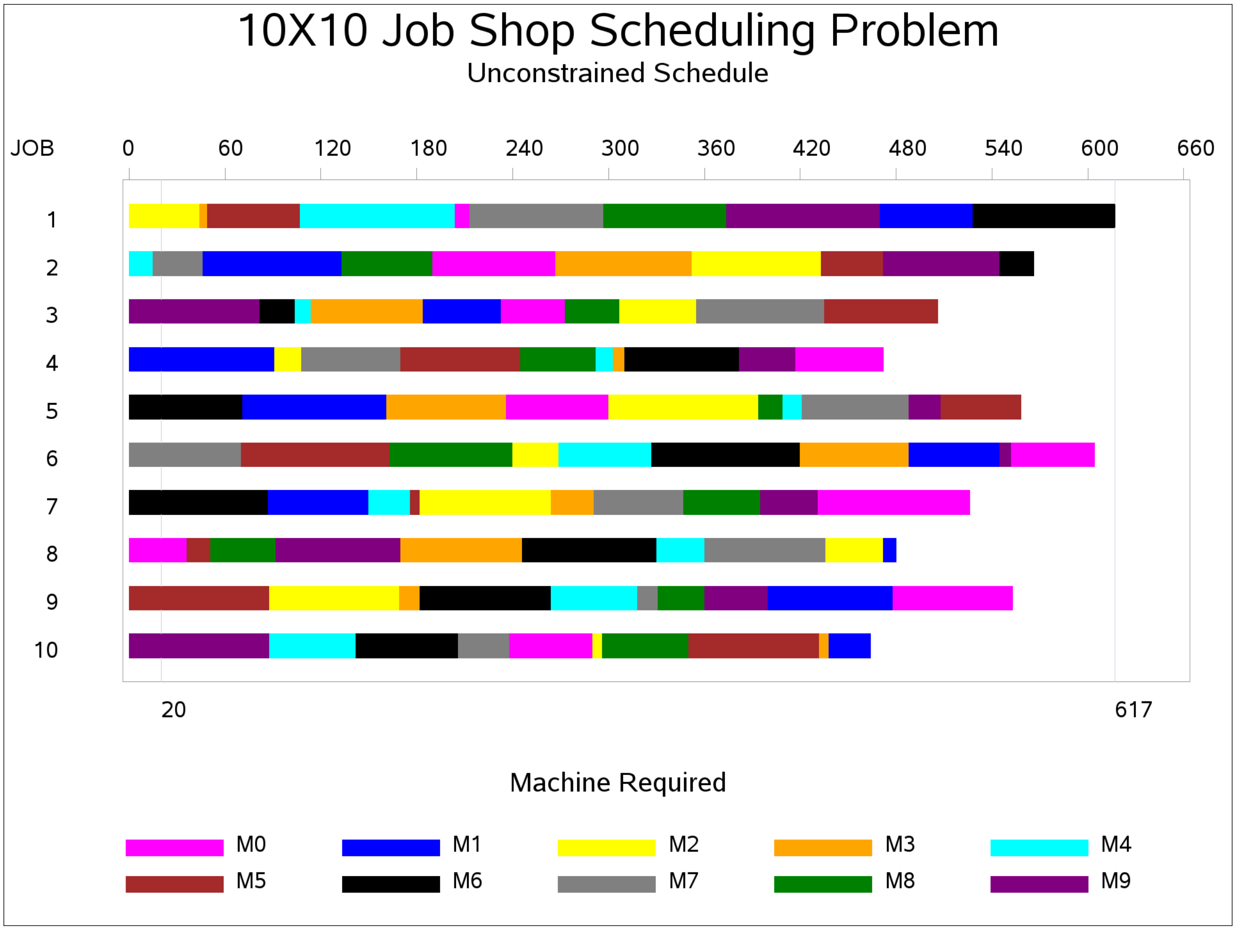
Each row in the DATALINES section specifies a job by 10 pairs of consecutive numbers. Each pair of numbers defines one task of the job, which represents the processing of a job on a machine. For each pair, the first number identifies the machine it executes on, and the second number is the duration. The order of the 10 pairs defines the sequence of the tasks for a job.

The following statements create the Activity data set actdata, which defines the activities, durations, and precedence constraints:

```
/* create the Activity data set */
data actdata (drop= i j);
   format _ACTIVITY_ $8. _SUCCESSOR_ $8.;
   set raw;
   _QTY_ = 1;
   i=mod(_n_-1,10)+1;
   j=int((_n_-1)/10)+1;
   _ACTIVITY_ = compress('J'||put(j,best.)||'P'||put(i,best.));
   JOB=j;
   TASK=i;
   if i LT 10 then
      _SUCCESSOR_ = compress('J'||put(j,best.)||'P'||put((i+1),best.));
   else
      _SUCCESSOR_ = ' ';
   output;
run;
```

Had there been sufficient machine capacity, the jobs could have been processed according to a schedule as shown in Output 3.11.1. The minimum makespan would be 617—the time it takes to complete Job 1.

Output 3.11.1 Gantt Chart: Schedule for the Unconstrained Problem



This schedule is infeasible when there is only a single instance of each machine. For example, at time period 20, the schedule requires two instances of each of the machines M6, M7, and M9.

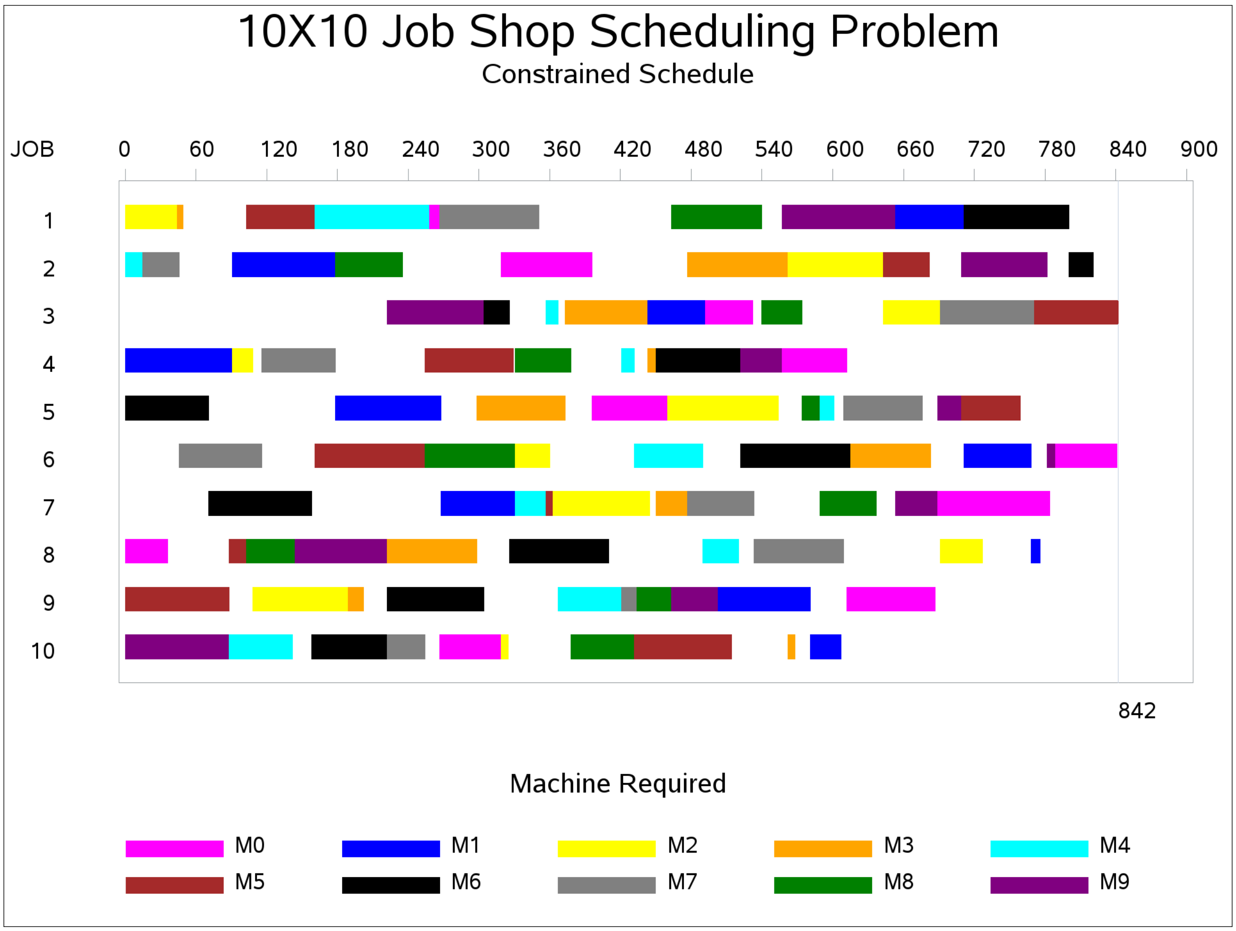
In order to solve the resource-constrained schedule, the CLP procedure is invoked by using the following statements:

```
proc clp domain=[0,842]
         actdata=actdata
         schedout=sched_jobshop
         dpr=50
         restarts=150
         showprogress;
   schedule finish=842 edgefinder nf=1 nl=1;
run;
```

The edge-finder algorithm is activated with the EDGEFINDER option in the SCHEDULE statement. In addition, the edge-finding extensions for detecting whether a job cannot be the first or cannot be the last to be processed on a particular machine are invoked with the NF= and NL= options, respectively, in the SCHEDULE statement. The default activity selection and activity assignment strategies are used. A restart heuristic is used as the look-back method to handle recovery from failures. The DPR= option specifies that a total restart be performed after encountering 50 failures, and the RESTARTS= option limits the number of restarts to 150.

The resulting 842-time-period schedule is displayed in Output 3.11.2. Each row represents a job. Each segment represents a task (the processing of a job on a machine), which is also coded according to the executing machine. The mapping of the coding is indicated in the legend. Note that no machine is used by more than one job at any point in time.

Output 3.11.2 Gantt Chart: Optimal Resource-Constrained Schedule



Example 3.12: Scheduling a Major Basketball Conference

Example 1.8 illustrated how you could use the CLP procedure to solve a single round-robin problem by modeling it as a scheduling CSP. This example illustrates an alternate way of modeling and solving a well-known double round-robin problem using the CLP procedure. This example is based on the work of Nemhauser and Trick (1998) and deals with scheduling the Atlantic Coast Conference (ACC) Men's Basketball games for the 1997–1998 season.

A temporally dense double round-robin (DDRR) for n teams is a double round-robin in which the $n(n-1)$ games are played over a minimal number of dates or time slots. If n is even, the number of slots is $2(n-1)$ and each team plays in every time slot. If n is odd, the number of slots is $2n$ and $(n-1)$ teams play in each time slot. In the latter case, each time slot has a team with a bye, and each team has two byes for the season.

The Atlantic Coast Conference (ACC) 1997–1998 men's basketball scheduling problem as described in Nemhauser and Trick (1998) and Henz (2001) is a DDRR that consists of the following nine teams with their abbreviated team name and team number shown in parentheses: Clemson (Clem 1), Duke (Duke 2), Florida State (FSU 3), Georgia Tech (GT 4), Maryland (UMD 5), University of North Carolina (UNC 6), NC State (NCSU 7), Virginia (UVA 8), and Wake Forest (Wake 9).

The general objective is to schedule the DDRR to span the months of January and February and possibly include a game in December or March or both. In general, each team plays twice a week—typically Wednesday and Saturday. Although the actual day might differ, these two time slots are referred to as the "weekday slot" and the "weekend slot." Since there are an odd number of teams, there is a team with a bye in each slot and four games in each slot, resulting in a schedule that requires 18 time slots or nine weeks. The last time slot must be a weekend slot, which implies the first slot is a weekday slot. The first slot, denoted slot 1, corresponds to the last weekday slot of December 1997, and the final slot, slot 18, corresponds to the first weekend slot of March 1998. Each team plays eight home games and eight away games, and has two byes.

In addition several other constraints must be satisfied. This example uses the following criteria employed by Nemhauser and Trick (1998) as presented by Henz (2001).

1. **Mirroring:** The dates are grouped into pairs $(r1, r2)$, such that each team gets to play against the same team in dates $r1$ and $r2$. Such a grouping is called a mirroring scheme. A separation of nine slots can be achieved by mirroring a round-robin schedule; although this separation is desirable, it is not possible for this problem.

   Nemhauser and Trick fix the mirroring scheme to

   $$m = (1,8),(2,9),(3,12),(4,13),(5,14),(6,15),(7,16),(10,17),(11,18)$$

   in order to satisfy the constraints that UNC and Duke play in time slots 11 and 18. (See criterion 9.)

2. **Initial and final home and away games:** Every team must play at home on at least one of the first three dates. Every team must play at home on at least one of the last three dates. No team can play away on both of the last two dates.

3. **Home/away/bye pattern:** No team can have more than two away games in a row. No team can have more than two home games in a row. No team can have more than three away games or byes in a row. No team can have more than four home games or byes in a row.

4. **Weekend pattern:** Of the nine weekends, each team plays four at home, four away, and has one bye.

5. **First weekends:** Each team must have home games or byes on at least two of the first five weekends.

6. **Rival matches:** Every team except FSU has a traditional rival. The rival pairs are Clem-GT, Duke-UNC, UMD-UVA, and NCSU-Wake. In the last date, every team except FSU plays against its rival, unless it plays against FSU or has a bye.

7. **Popular matches in February:** The following pairings must occur at least once in dates 11 to 18: Duke-GT, Duke-Wake, GT-UNC, UNC-Wake.

8. **Opponent sequence:** No team plays in two consecutive away dates against Duke and UNC. No team plays in three consecutive dates against Duke, UNC, and Wake (independent of the home or away status).

9. **Idiosyncrasies:** UNC plays its rival Duke in the last date and in date 11. UNC plays Clem in the second date. Duke has a bye in date 16. Wake does not play home in date 17. Wake has a bye in the first date. Clem, Duke, UMD and Wake do not play away in the last date. Clem, FSU, and GT do not play away in the first date. Neither FSU nor NCSU has a bye in last date. UNC does not have a bye in the first date.

Previous work for solving round-robin problems, including that of Nemhauser and Trick (1998) and Henz (2001), have used a general three-phase framework for finding good schedules.

1. pattern generation
2. pattern set generation
3. timetable generation

A pattern is a valid sequence of home, away, and bye games for a given team for the entire season. For example, the following is a valid pattern:

A H B A H H A H A A H B H A A H H A

For this example, patterns that satisfy criterion 1 through criterion 5 and some constraints in criterion 9 are generated using the CLP procedure with the SAS macro %PATTERNS.

```
/****************************************************************/
/* First, find all possible patterns. Consider only time        */
/* constraints at this point. A pattern should be suitable      */
/* for any team. Do not consider individual teams yet.          */
/****************************************************************/
%macro patterns();

proc clp out=all_patterns findall;
   /* For date 1 to 18. */
   %do j = 1 %to 18;
      var h&j = [0, 1]; /* home */
      var a&j = [0, 1]; /* away */
      var b&j = [0, 1]; /* bye */

      /* A team is either home, away, or bye. */
      lincon h&j + a&j + b&j=1;
   %end;

   /*-----------------------------------------------------------*/
   /* Criterion 1 - Mirroring Scheme                            */
   /*-----------------------------------------------------------*/
   /* The dates are grouped into pairs (j, j1), such that each  */
   /* team plays the same opponent on dates j and j1.           */
   /* A home game on date j will be an away game on date j1     */
   %do j = 1 %to 18;
      %do j1 = %eval(&j+1) %to 18;
         %if ( &j=1 and &j1=8 ) or ( &j=2 and &j1=9 ) or
             ( &j=3 and &j1=12 ) or ( &j=4 and &j1=13 ) or
             ( &j=5 and &j1=14 ) or ( &j=6 and &j1=15 ) or
```

```
          ( &j=7 and &j1=16 ) or ( &j=10 and &j1=17 ) or
          ( &j=11 and &j1=18 ) %then
         lincon h&j = a&j1, a&j = h&j1, b&j = b&j1;;
   %end;
%end;

/*-------------------------------------------------------------*/
/* Criterion 2 - Initial and Final Home and Away Games        */
/*-------------------------------------------------------------*/
/* Every team must play home on at least one of the first three dates. */
lincon h1 + h2 + h3 >= 1;

/* Every team must play home on at least one of the last three dates. */
lincon h16 + h17 + h18 >= 1;

/* No team can play away on both last two dates. */
lincon a17 + a18 < 2;

/*-------------------------------------------------------------*/
/* Criterion 3 - Home/Away/Bye Pattern                         */
/*-------------------------------------------------------------*/
%do j = 1 %to 16;
   /* No team can have more than two away matches in a row.*/
   lincon a&j + a%eval(&j+1) + a%eval(&j+2) < 3;
   /* No team can have more than two home matches in a row.*/
   lincon h&j + h%eval(&j+1) + h%eval(&j+2) < 3;
%end;

/* No team can have more than three away matches or byes in a row.*/
%do j = 1 %to 15;
   lincon a&j + b&j + a%eval(&j+1) + b%eval(&j+1) + a%eval(&j+2)
          + b%eval(&j+2) + a%eval(&j+3) + b%eval(&j+3) < 4;
%end;

/* No team can have more than four home matches or byes in a row.*/
%do j = 1 %to 14;
    lincon h&j + b&j + h%eval(&j+1) + b%eval(&j+1) + h%eval(&j+2)
           + b%eval(&j+2) + h%eval(&j+3) + b%eval(&j+3) + h%eval(&j+4)
           + b%eval(&j+4) < 5;
%end;

/*-------------------------------------------------------------*/
/* Criterion 4 - Weekend Pattern                               */
/*-------------------------------------------------------------*/
/* Each team plays four weekends at home. */
lincon 0 %do j = 2 %to 18 %by 2; +h&j %end; =4;
/* Each team plays four weekends away. */
lincon 0 %do j = 2 %to 18 %by 2; +a&j %end; =4;
/* Each team has 1 weekend with a bye  */
lincon 0 %do j = 2 %to 18 %by 2; +b&j %end; =1;

/*-------------------------------------------------------------*/
/* Criterion 5 - First Weekends                                */
/*-------------------------------------------------------------*/
```

```
   /* Each team must have home games or byes on at least two     */
   /* of the first five weekends.                                */
   lincon 0 %do j = 2 %to 10 %by 2; + h&j + b&j %end; >=2;

   /*------------------------------------------------------------*/
   /* Criterion 9 - (Partial)                                    */
   /*------------------------------------------------------------*/
   /* The team with a bye in date 1 does not play away on the    */
   /*    last date or home in date 17 (Wake)                     */
   /* The team with a bye in date 16 does not play away in       */
   /*    date 18 (Duke)                                          */
   lincon b1 + a18 < 2, b1 + h17 < 2, b16 + a18 < 2;

run;

%mend;

%patterns;
```

The %PATTERNS macro generates 38 patterns. The next step is to find a subset of patterns with cardinality equal to the number of teams that would collectively support a potential assignment to all of the teams. For example, each of the 18 time slots must correspond to four home games, four away games, and one bye. Furthermore, pairs of patterns that do not support a potential meeting date between the two corresponding teams are excluded. The following %PATTERN_SETS macro uses the CLP procedure with the preceding constraints to generate 17 possible pattern sets.

```
/*****************************************************************/
/* Determine all possible "pattern sets" considering only time   */
/* constraints.                                                  */
/* Individual teams are not considered at this stage.           */
/* xi - binary variable indicates pattern i is in pattern set    */
/*****************************************************************/

%macro pattern_sets();

data _null_;
   set all_patterns;
   %do i=1 %to 38;
      if _n_=&i then do;
         %do j=1 %to 18;
            call symput("h&i._&j", put(h&j,best.));
            call symput("a&i._&j", put(a&j,best.));
            call symput("b&i._&j", put(b&j,best.));
         %end;
      end;
   %end;
run;

proc clp out=pattern_sets findall;
   /* xi=1 if pattern i belongs to pattern set */
   var (x1-x38)= [0, 1];

   /* Exactly nine patterns per patterns set */
```

```
   lincon 0 %do i = 1 %to 38; + x&i %end;=9;

   /* time slot constraints */
   %do j = 1 %to 18;
      /* Four home games per time slot */
      lincon 0 %do i = 1 %to 38; + &&h&i._&j*x&i %end; =4;
      /* Four away games per time slot */
      lincon 0 %do i = 1 %to 38; + &&a&i._&j*x&i %end; =4;
      /* One bye per time slot */
      lincon 0 %do i = 1 %to 38; + &&b&i._&j*x&i %end; =1;
   %end;

   /* Exclude pattern pairs that do not support a meeting date */
   %do i = 1 %to 38;
      %do i1 = %eval(&i+1) %to 38;
         %let count=0;
         %do j=1 %to 18;
            %if ( (&&h&i._&j=0 or &&a&i1._&j=0) and
                  (&&a&i._&j=0 or &&h&i1._&j=0)) %then %do;
               %let count=%eval(&count+1);
            %end;
         %end;
         %if (&count=18) %then %do;
            lincon x&i+x&i1<=1;
         %end;
      %end;
   %end;
run;

%mend;

%pattern_sets;
```

The %PATTERN_SETS macro generates 17 pattern sets. The final step is to add the individual team constraints and match up teams to the pattern set in order to come up with a schedule for each team. The schedule for each team indicates the opponent for each time slot (0 for a bye) and whether it corresponds to a home game, away game, or a bye.

The following SAS macro %TIMETABLE uses the pattern set index as a parameter and invokes the CLP procedure with the individual team constraints to determine the team schedule.

```
/*****************************************************************/
/* Assign individual teams to pattern set k                      */
/* Teams: 1 Clem, 2 Duke, 3 FSU, 4 GT, 5 UMD, 6 UNC, 7 NCSU, 8 UVA,  */
/*        9 Wake                                                 */
/*****************************************************************/
%macro timetable(k);

proc clp out=ACC_ds_&k varselect=minrmaxc findall;

   %do j = 1 %to 18;
      /*  alpha(i,j): Team i's opponent on date j ( 0 = bye ). */
      %do i = 1 %to 9;
```

```
      var alpha&i._&j = [0, 9];
   %end;

   /* Timetable constraint 1 */
   /* Opponents in a time slot must be distinct */
   alldiff ( %do i = 1 %to 9; alpha&i._&j %end; );

   /* Timetable constraint 2 */
   %do i = 1 %to 9;
      %do i1 = 1 %to 9;
         /* indicates if teams i and i1 play in time slot j */
         var X&i._&i1._&j = [0, 1];
         reify X&i._&i1._&j: (alpha&i._&j = &i1);

         /* team i plays i1 iff team i1 plays i */
         %if (&i1 > &i ) %then %do;
            lincon X&i._&i1._&j = X&i1._&i._&j;
         %end;
      %end;
   %end;
%end;


/* Mirroring Scheme at team level.                          */
/* The dates are grouped into pairs (j, j1) such that each */
/* team plays the same opponent in dates j and j1.         */
/* One of these should be a home game for each team.       */
%do i = 1 %to 9;
   %do j = 1 %to 18;
      %do j1 = %eval(&j+1) %to 18;
         %if ( &j=1 and &j1=8 ) or ( &j=2 and &j1=9 ) or
             ( &j=3 and &j1=12 ) or ( &j=4 and &j1=13 ) or
             ( &j=5 and &j1=14 ) or ( &j=6 and &j1=15 ) or
             ( &j=7 and &j1=16 ) or ( &j=10 and &j1=17 ) or
             ( &j=11 and &j1=18 ) %then %do;
            lincon alpha&i._&j=alpha&i._&j1,
            /* H and A are matrices that indicate home */
            /* and away games                          */
            H&i._&j=A&i._&j1,
            H&i._&j1=A&i._&j;
         %end;
      %end;
   %end;
%end;

/* Timetable constraint 3 */
/* Each team plays every other team twice */
%do i = 1 %to 9;
   %do i1 = 1 %to 9;
      %if &i1 ne &i %then %do;
         lincon 0 %do j = 1 %to 18; + X&i._&i1._&j %end; = 2;
      %end;
   %end;
%end;
```

```
/* Timetable constraint 4 */
/* Teams do not play against themselves */
%do j = 1 %to 18;
   %do i = 1 %to 9;
      lincon alpha&i._&j<>&i;
      lincon X&i._&i._&j = 0;  /* redundant */
   %end;
%end;

/* Timetable constraint 5 */
/* Setup Bye Matrix */
/* alpha&i._&j=0 means team &i has a bye on date &j. */
%do j = 1 %to 18;
   %do i = 1 %to 9;
      var B&i._&j = [0, 1]; /*Bye matrix*/
      reify B&i._&j: ( alpha&i._&j = 0 );
   %end;
%end;

/* Timetable constraint 6 */
/* alpha&i._&j=&i1 implies teams &i and &i1 play on date &j */
/* It must be a home game for one, away game for the other  */
%do j = 1 %to 18;
   %do i = 1 %to 9;
      %do i1 = 1 %to 9;
         /* reify control variables.*/
         var U&i._&i1._&j = [0, 1] V&i._&i1._&j = [0, 1];

         /* if &i is home and &i1 is away. */
         reify U&i._&i1._&j: ( H&i._&j + A&i1._&j = 2);
         /* if &i1 is home and &i is away. */
         reify V&i._&i1._&j: ( A&i._&j + H&i1._&j = 2);

         /* Necessary condition if &i plays &i1 on date j  */
         lincon X&i._&i1._&j <= U&i._&i1._&j + V&i._&i1._&j;
      %end;
   %end;
%end;

/* Timetable constraint 7 */
/* Each team must be home, away or have a bye on a given date */
%do j = 1 %to 18;
   %do i = 1 %to 9;
      /* Team &i is home (away) at date &j. */
      var H&i._&j = [0, 1] A&i._&j = [0, 1];
      lincon H&i._&j + A&i._&j + B&i._&j = 1;
   %end;
%end;

%do i = 1 %to 9;
   %do i1 = %eval(&i+1) %to 9;

      /* Timetable constraint 8 */
```

```
      /*-------------------------------------------------------*/
      /* Criterion 6 - Rival Matches                            */
      /*-------------------------------------------------------*/
      /* The final weekend is reserved for 'rival games' */
      /* unless the team plays FSU or has a bye          */
      %if ( &i=1 and &i1=4 ) or ( &i=2 and &i1=6 ) or
          ( &i=5 and &i1=8 ) or ( &i=7 and &i1=9 ) %then %do;
         lincon X&i._&i1._18 + B&i._18 + X&i._3_18 = 1;

         /* redundant */
         lincon X&i1._&i._18 + B&i1._18 + X&i1._3_18 = 1;
      %end;

      /* Timetable constraint 9 */
      /*-------------------------------------------------------*/
      /* Criterion 7 - Popular Matches                          */
      /*-------------------------------------------------------*/
      /* The following pairings are specified to occur at      */
      /* least once in February.                                */
      %if ( &i=2 and &i1=4 ) or ( &i=2 and &i1=9 ) or
          ( &i=4 and &i1=6 ) or ( &i=6 and &i1=9 ) %then %do;
         lincon 0 %do j = 11 %to 18; + X&i._&i1._&j  %end; >= 1;

         /* redundant */
         lincon 0 %do j = 11 %to 18; + X&i1._&i._&j %end; >= 1;
      %end;
   %end;
%end;

/* Timetable constraint 10 */
/*-------------------------------------------------------*/
/* Criterion 8 - Opponent Sequence                        */
/*-------------------------------------------------------*/
%do i = 1 %to 9;
   /* No team plays two consecutive away dates against   */
   /* Duke (2) and UNC (6)                               */
   %do j = 1 %to 17;
      var Q&i._26_&j = [0, 1] P&i._26_&j = [0, 1];
      reify Q&i._26_&j: ( X&i._2_&j +  X&i._6_&j = 1 );
      reify P&i._26_&j: ( X&i._2_%eval(&j+1) +  X&i._6_%eval(&j+1) =1 );
      lincon Q&i._26_&j + A&i._&j + P&i._26_&j + A&i._%eval(&j+1) < 4;
   %end;

   /* No team plays three consecutive dates against      */
   /* Duke(2), UNC(6) and Wake(9).                       */
   %do j = 1 %to 16;
      var L&i._269_&j = [0, 1] M&i._269_&j = [0, 1]
          N&i._269_&j = [0, 1];
      reify L&i._269_&j: ( X&i._2_&j +  X&i._6_&j +  X&i._9_&j = 1 );
      reify M&i._269_&j: ( X&i._2_%eval(&j+1) + X&i._6_%eval(&j+1) +
                           X&i._9_%eval(&j+1) =1 );
      reify N&i._269_&j: ( X&i._2_%eval(&j+2) + X&i._6_%eval(&j+2) +
                           X&i._9_%eval(&j+2) =1 );
      lincon L&i._269_&j + M&i._269_&j + N&i._269_&j  < 3;
```

```
      %end;
   %end;


   /* Timetable constraint 11 */
   /*------------------------------------------------------*/
   /* Criterion 9 - Idiosyncratic Constraints              */
   /*------------------------------------------------------*/
   /* UNC plays Duke in date 11 and 18 */
   lincon alpha6_11 = 2 ;
   lincon alpha6_18 = 2 ;
   /* UNC plays Clem in the second date. */
   lincon alpha6_2 = 1 ;
   /* Duke has a bye in date 16. */
   lincon B2_16 = 1 ;
   /* Wake does not play home in date 17. */
   lincon H9_17 = 0 ;
   /* Wake has a bye in the first date. */
   lincon B9_1 = 1 ;
   /* Clem, Duke, UMD and Wake do not play away in the last date. */
   lincon A1_18 = 0 ;
   lincon A2_18 = 0 ;
   lincon A5_18 = 0 ;
   lincon A9_18 = 0 ;
   /* Clem, FSU, and GT do not play away in the first date. */
   lincon A1_1 = 0 ;
   lincon A3_1 = 0 ;
   lincon A4_1 = 0 ;
   /* FSU and NCSU do not have a bye in the last date. */
   lincon B3_18 = 0 ;
   lincon B7_18 = 0 ;
   /* UNC does not have a bye in the first date. */
   lincon B6_1 = 0 ;

   /* Timetable constraint 12 */
   /*------------------------------------------------------*/
   /* Match teams with patterns.                           */
   /*------------------------------------------------------*/
   %do i = 1 %to 9; /* For each team */
      var p&i=[1,9];
      %do j=1 %to 18; /* For each date */
         element ( p&i, (&&col&k._h_&j), H&i._&j )
                 ( p&i, (&&col&k._a_&j), A&i._&j )
                 ( p&i, (&&col&k._b_&j), B&i._&j );
      %end;
   %end;
run;

%mend;

/**************************************************************/
/* Try all possible pattern sets to find all valid schedules. */
/**************************************************************/
```

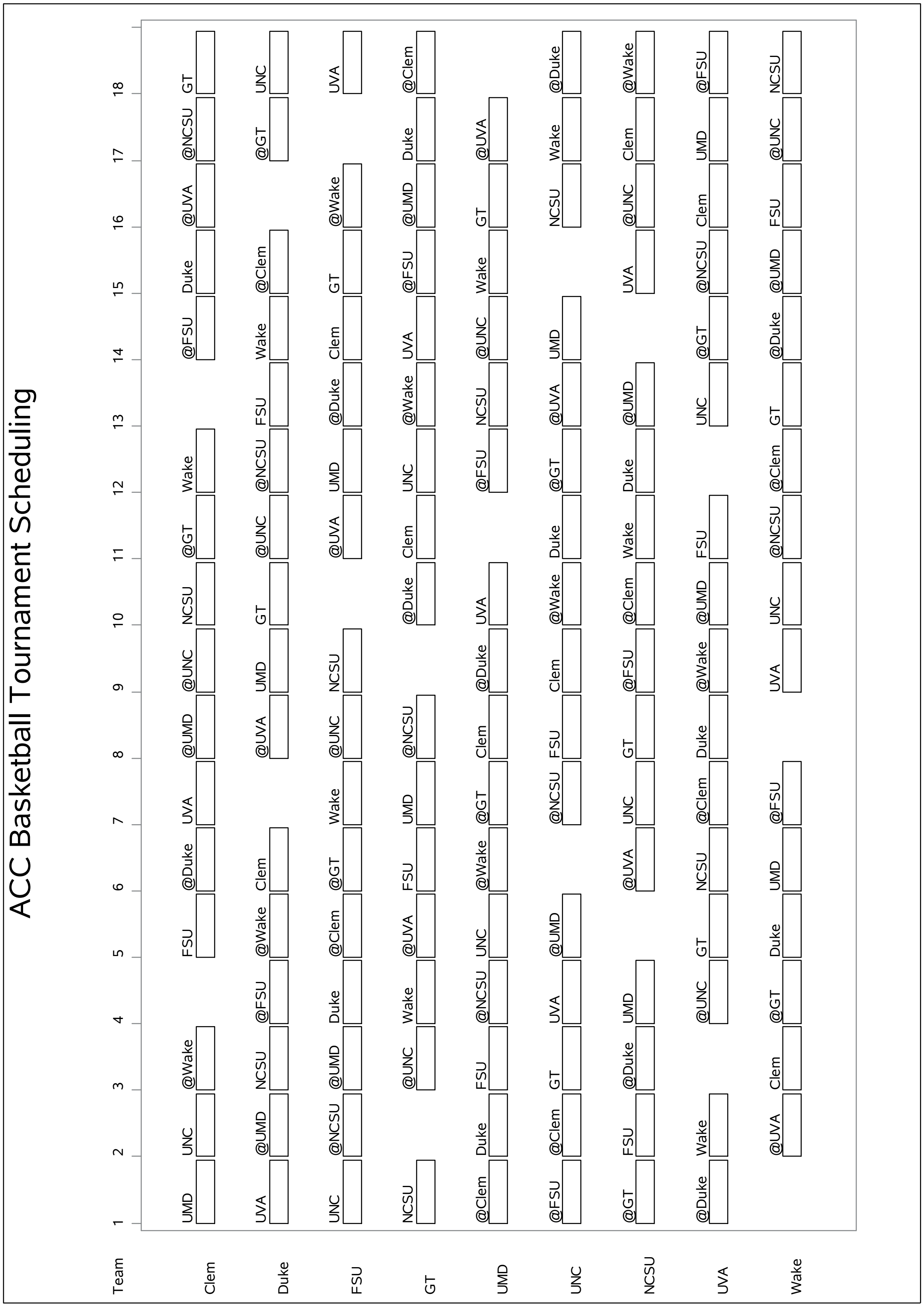
```
%macro find_schedules;

proc transpose data=pattern_sets out=trans_good; run;

data _temp;
   set trans_good;
   set all_patterns;
run;

proc sql noprint;
   %do k = 1 %to 17; /* For each pattern */
      %do j=1 %to 18; /* For each date */
         select h&j into :col&k._h_&j
            separated by ',' from _temp where col&k=1;
         select a&j into :col&k._a_&j
            separated by ',' from _temp where col&k=1;
         select b&j into :col&k._b_&j
            separated by ',' from _temp where col&k=1;
      %end;
   %end;
run;


data all; run;

%do k = 1 %to 17; /* For each pattern set */
   %timetable(k=&k);

   data all;
      set all ACC_ds_&k;
   run;
%end;

data all;
   set all;
   if _n_=1 then delete;
run;

%mend;

%find_schedules;
```

The %FIND_SCHEDULES macro invokes the %TIMETABLE macro for each of the 17 pattern sets and generates 179 possible schedules, including the one that the ACC eventually used, which is displayed in Output 3.12.1.

Example 3.13: 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:

Output 3.12.1 ACC Basketball Tournament Schedule



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

A BIBD is therefore specified by its parameters (v, b, r, k, λ). 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 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, the condition that each block contains exactly k objects (there are k 1s per column) can be specified by the following constraints:

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

To enforce the final condition that every two distinct objects occur together in exactly λ blocks (equivalently, that the scalar product of every pair of rows equal λ), 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 reified 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 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 lexicographical 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 lexicographical 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, λ), 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 clp out=&out(keep=x:) domain=[0,1] varselect=FIFO;
      /* Decision variables: */
      /* Decision variable Xi_c = 1 iff object i occurs in block c. */
      var (
           %do i=1 %to &v;
              x&i._1-x&i._&b.
           %end;
          ) = [0,1];

      /* Mandatory constraints: */
      /* (i) Each object occurs in exactly r blocks. */
      %let q = %eval(&b.-&r.);  /* each row has &q 0s and &r 1s */
      %do i=1 %to &v;
         gcc( x&i._1-x&i._&b. ) = ((0,0,&q.) (1,0,&r.));
      %end;

      /* (ii) Each block contains exactly k objects. */
      %let h = %eval(&v.-&k.);  /* each column has &h 0s and &k 1s */
      %do c=1 %to &b;
         gcc(
             %do i=1 %to &v;
```

```
                x&i._&c.
             %end;
            ) = ((0,0,&h.) (1,0,&k.));
      %end;

      /* (iii) Every pair of objects occurs in exactly lambda blocks. */
      %let t = %eval(&b.-&lambda.);
      %do i=1 %to %eval(&v.-1);
         %do j=%eval(&i.+1) %to &v;
            /* auxiliary variable p_i_j_c =1 iff both i and j occur in c */
            var ( p&i._&j._1-p&i._&j._&b. ) = [0,1];
            %do c=1 %to &b;
               reify p&i._&j._&c.: (x&i._&c. + x&j._&c. = 2);
            %end;

            gcc(p&i._&j._1-p&i._&j._&b.) = ((0,0,&t.) (1,0,&lambda.));
         %end;
      %end;

      /* Symmetry breaking constraints: */
      /* Break row symmetry via lexicographic ordering constraints. */
      %do i = 2 %to &v.;
         %let i1 = %eval(&i.-1);
         lexico( (x&i._1-x&i._&b.) LEX_LT (x&i1._1-x&i1._&b.) );
      %end;

      /* Break column symmetry via lexicographic ordering constraints. */
      %do c = 2 %to &b.;
         %let c1 = %eval(&c.-1);
         lexico( ( %do i = 1 %to &v.;
                      x&i._&c.
                   %end; )
                 LEX_LE
                 ( %do i = 1 %to &v.;
                      x&i._&c1.
                   %end; ) );
      %end;
   run;
   %put &_orclp_;
%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 3.13.1.

Output 3.13.1 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 3.14: 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 thirteen hosts is assumed (boats 1 through 12 and 14). The total capacities and crew sizes of the boats are shown in Figure 3.3.

Figure 3.3 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:

- Alldifferent 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 the CLP procedure to define the variables, specify the constraints, and solve the problem.

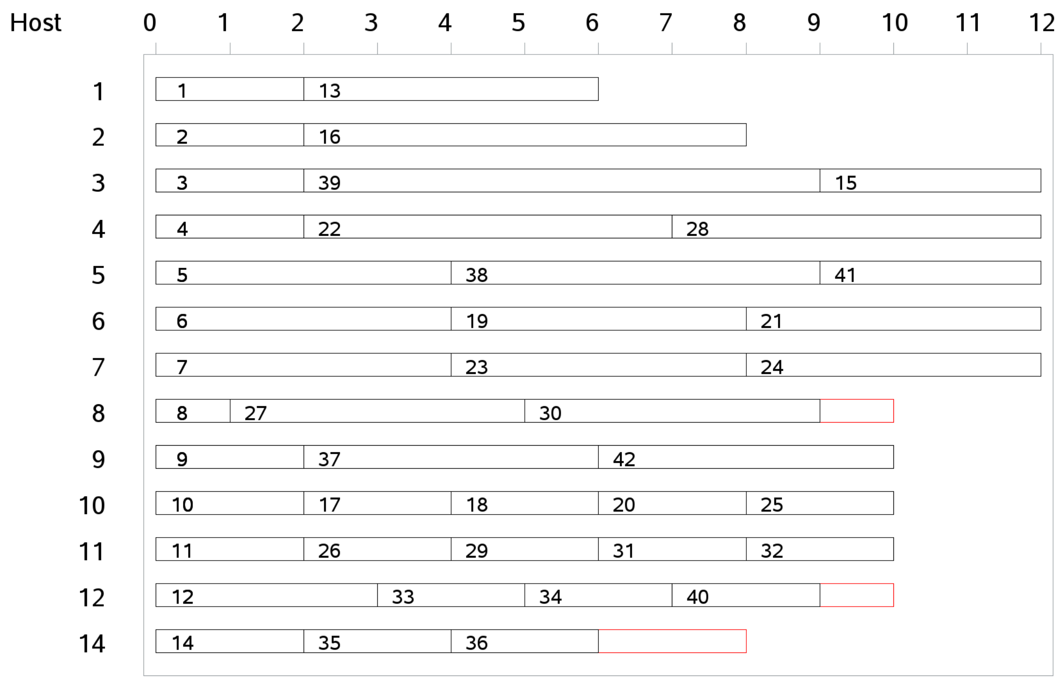
```
%let rounds=2;
%let numhosts=13;

%macro ppp;
   proc sql noprint;
      select count(*) into :numboats from capacities;
      select max(capacity) into :maxcap from capacities;
      %do i = 0 %to &maxcap;
         select count(*) into :numclass_&i from capacities where capacity = &i;
      %end;
      select crewsize, spareCapacity into
         :crewsize_1-:crewsize_%scan(&numboats,1),
         :cap_1-:cap_%scan(&numboats,1) from capacities order by seqno;
   quit;

   proc clp out=out varselect=FIFO;
      /* assume first &numhosts boats are hosts */
      /* process each round in turn */
      %do t = 1 %to &rounds;
         %do i = &numhosts+1 %to &numboats;
            /* boat i assigned host value for round t */
            var x_&i._&t = [1,&numhosts];
         %end;
         %do h = 1 %to &numhosts;
            var L_&h._&t = [0,&&cap_&h]; /* load of host boat */
         %end;
      %end;

      %do i = &numhosts+1 %to &numboats;
         /* assign different host each round */
         alldiff (x_&i._1-x_&i._&rounds);
      %end;

      %do t = 1 %to &rounds;
         %do i = &numhosts+1 %to &numboats-1;
            /* boat i assigned host value for round t */
            %do j = &i+1 %to &numboats;
               var m_&i._&j._&t = [0,1];
               reify m_&i._&j._&t : (x_&i._&t = x_&j._&t);
            %end;
         %end;
      %end;

      %do i = &numhosts+1 %to &numboats-1;
         %do j = &i+1 %to &numboats;
            lincon 1 >= 0
            %do t = 1 %to &rounds;
               + m_&i._&j._&t
            %end;
            ;
         %end;
      %end;
```

```
      /* honor capacities */
      %do t = 1 %to &rounds;
         PACK((
         %do i = &numhosts+1 %to &numboats;
            x_&i._&t
         %end;
         ) (
         %do i = &numhosts+1 %to &numboats;
            &&crewsize_&i
         %end;
         ) (
         %do h = 1 %to &numhosts;
            L_&h._&t
         %end;
         ));
      %end;

      /* break symmetries */
      %do t = 1 %to &rounds-1;
         lincon x_%scan(&numboats,1)_&t < x_%scan(&numboats,1)_%eval(&t+1);
      %end;

   run;
%mend ppp;

%ppp;
```

The two charts in Output 3.14.1 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.

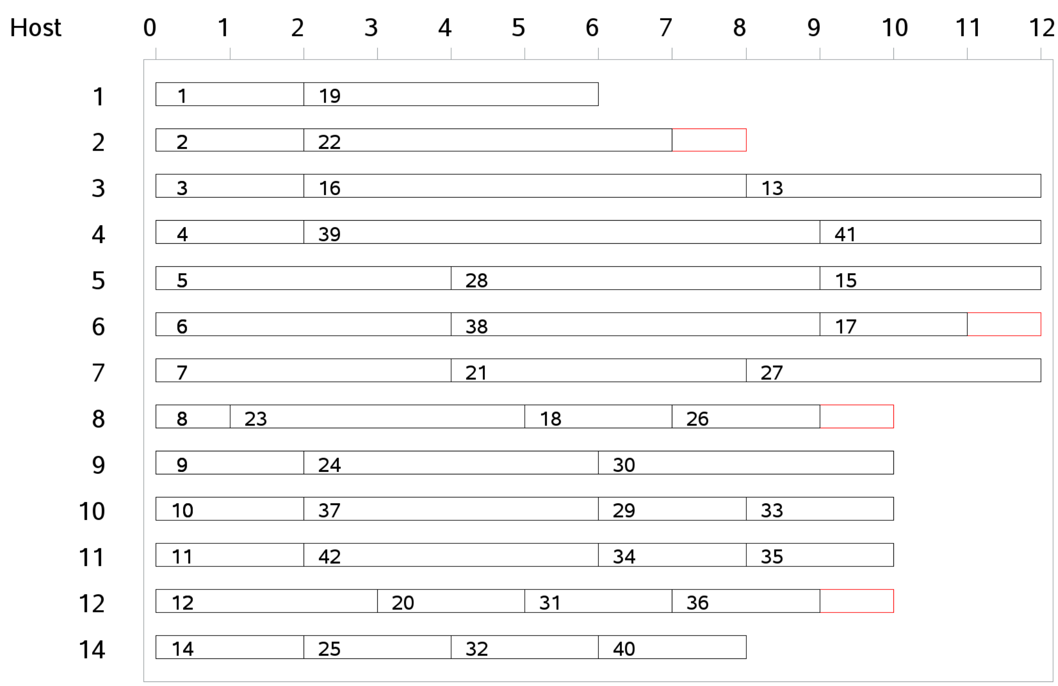
Output 3.14.1 Gantt Chart: Boat Schedule by Round

Schedule for Round 1



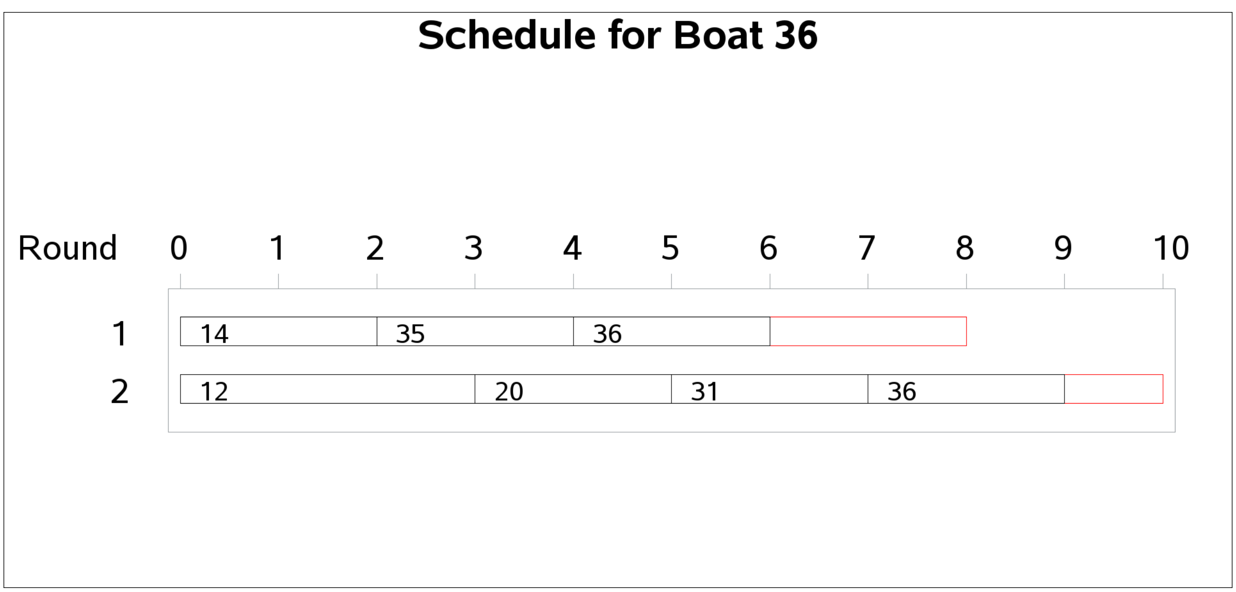
Output 3.14.1 *continued*

Schedule for Round 2

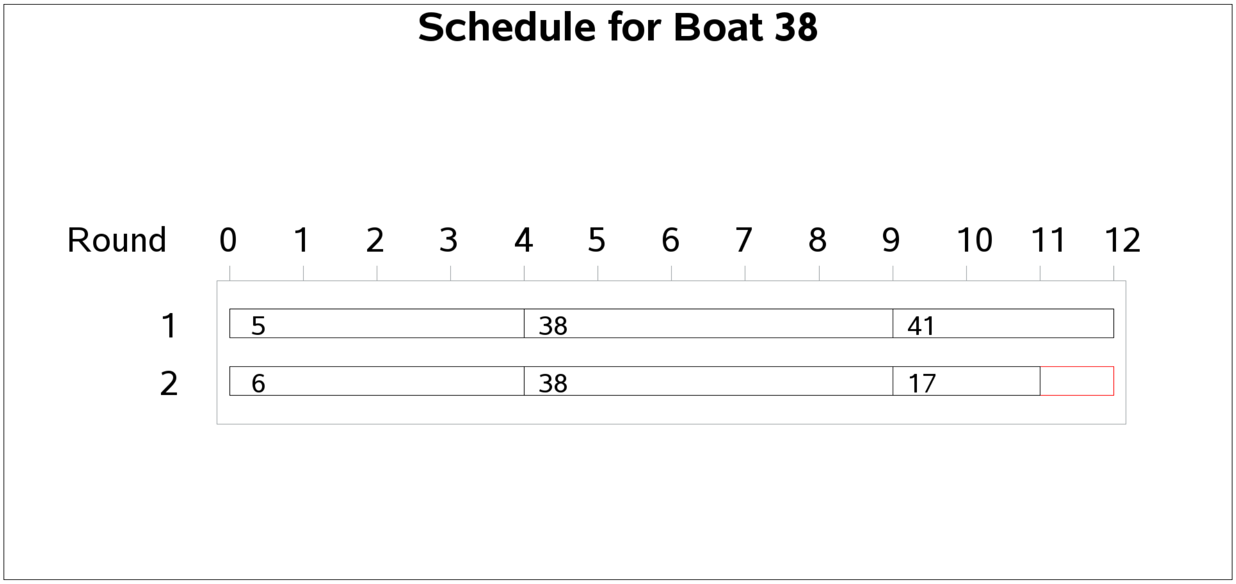


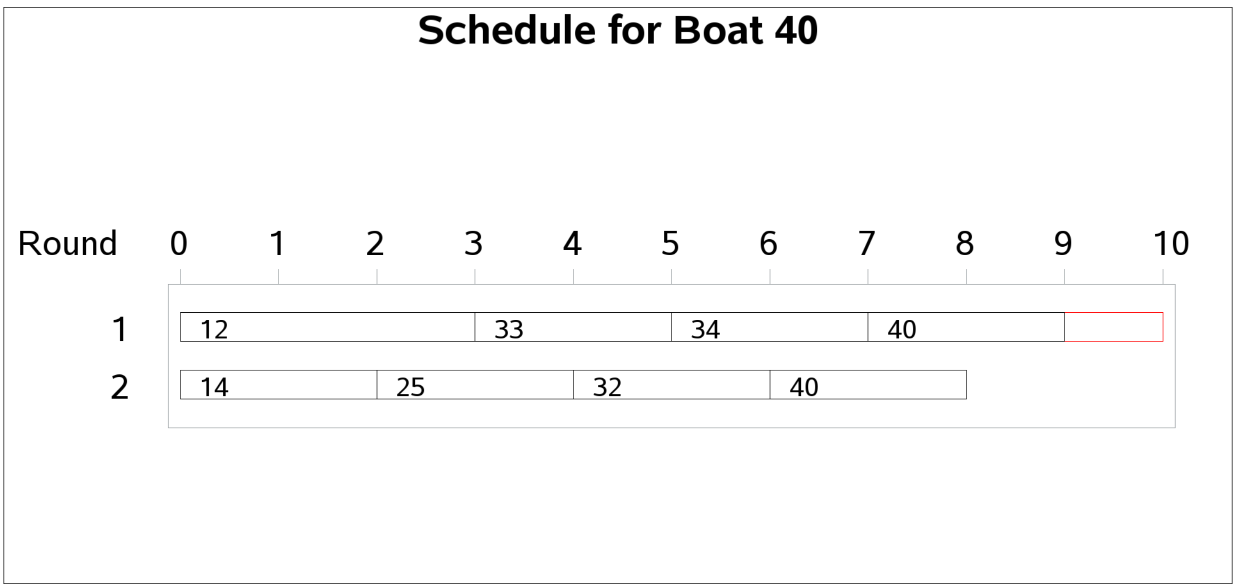
The charts in Output 3.14.2 break down the assignments by boat number for selected boats.

Output 3.14.2 Gantt Chart: Host Boat Schedule by Round

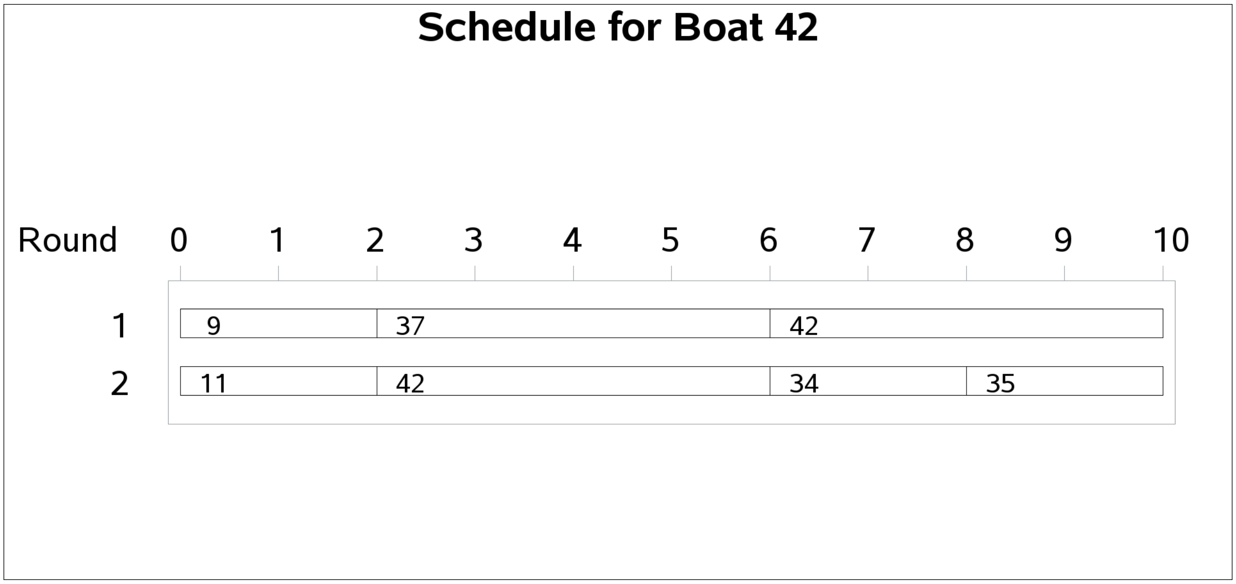


Output 3.14.2 *continued*





Output 3.14.2 *continued*



Example 3.15: Resource-Constrained Project Scheduling Problem with Time Windows

The general RCPSP/max (resource-constrained project scheduling problem with time windows) problem can be described as a single project that consists of a set of tasks (activities) that have to be scheduled. Each task requires time and renewable resources in order to be scheduled. In general, tasks are interrelated by two types of constraints:

- General temporal constraints restrict minimal and maximal time lags between two tasks. The temporal constraint between Task *i* and Task *j* has the form

  $\text{start\_time}[j] - \text{start\_time}[i] \in [\text{T}^{\text{min}}_{i,j}, \text{T}^{\text{max}}_{i,j}]$

  where $\text{T}^{\text{min}}_{i,j}$ and $\text{T}^{\text{max}}_{i,j}$ are the minimum and maximum time lags between tasks *i* and *j*.

- Resource constraints enforce the requirement that resource capacities are not exceeded at any time.

The objective of RCPSP/max is to find a feasible schedule that satisfies all temporal and resource constraints and that minimizes the makespan (duration) of the project.

Scheduling with Calendars

In many real-world scheduling problems, resources might not be available at all times, or they might have only partial capacities at some predetermined time periods (for example, because of weekends, holidays, or periods of machine maintenance). For each resource, a calendar indicates the time periods during which the resource is not available or has only partial capacity. The resource calendars impose additional restrictions on the problems.

This example illustrates the use of the cumulative constraint to model resource calendars in solving a single resource-constrained project scheduling problem.

This example considers a simple case of RCPSP/max that uses resource calendars and contains 12 uninterruptible tasks, each with fixed duration and demand. These tasks need to be scheduled on a single renewable resource that has a capacity of eight units. In addition, the resource has only half its capacity available during time period 5 and is unavailable in time period 6.

To simplify the specification, this example considers only the standard finish-to-start precedence relationships between two tasks. That is, Task *i* immediately precedes Task *j*, which implies:

$$\text{start\_time}[j] - \text{start\_time}[i] \geq \text{duration}[i]$$

More general forms of temporal relationships that involve nonstandard relationships and lags can be easily formulated using linear constraints.

Table 3.14 shows the durations, demands, and precedence relations of the tasks.

Table 3.14 Durations, Demands, and Successors of Tasks

Task	Duration	Demand	Successors
1	1	4	4
2	2	2	5
3	2	3	
4	6	3	
5	3	2	
6	6	3	12
7	1	1	8, 9, 10
8	3	2	11
9	3	2	12
10	4	1	12
11	2	2	12
12	4	2	

In order to accommodate the resource calendar, two artificial tasks, Task 13 and Task 14, are introduced, each with a duration of 1. Task 13 starts at time period 5 and uses four units of the resource, leaving half the capacity for other tasks during that period. Similarly, Task 14 starts at time period 6 and consumes the full capacity, thus making the corresponding resource unavailable to other tasks during that period.

Two additional tasks, *S* and *T*, are introduced to represent the beginning and completion of the project, respectively. The durations of both *S* and *T* are 0, and they do not require any of the resources. Note that the start time and the end time are equal for these two tasks. To simplify the notation, you can use S as the start-time variable of the project and T as the end-time variable. The makespan of the project can be expressed as T – S. Without loss of generality, the start time of the project is assumed to be 0, so that the objective of the problem is to minimize T.

There are two ways to declare start-time variables and specify temporal constraints:

- VARIABLE and LINCON statements
- Constraint data set

This example illustrates using the Constraint data set, which is created by merging three data sets. The StartTime data set, shown in Output 3.15.1, specifies the lower and upper bounds of the start-time variables for the real and artificial tasks. Note that the domains of artificial variables Start13, Start14, and S are fixed. A lower bound on variable T is determined by considering the (Task 6, Task 12) subpath, and an upper bound on T is determined by aggregating the durations of Task 1 through Task 14. The lower and upper bounds for the other variables are temporarily set to the lower bound of S and the upper bound of T, respectively, for convenience. Tighter bounds will be determined by the precedence constraints.

Output 3.15.1 StartTime Data Set

Obs	Start1	Start2	Start3	Start4	Start5	Start6	Start7	Start8	Start9	Start10	Start11	Start12
1	0	0	0	0	0	0	0	0	0	0	0	0
2	40	40	40	40	40	40	40	40	40	40	40	40
3	.	.	.	.	.	.	.	.	.	.	.	.

Obs	Start13	Start14	S	T	_TYPE_	_RHS_
1	.	.	.	10	LOWERBD	.
2	.	.	.	40	UPPERBD	.
3	5	6	0	.	FIXED	.

The Precedence data set shown in Output 3.15.2 specifies linear constraints that represent the precedence relations between tasks.

Output 3.15.2 Precedence Constraints

Obs	S	Start1	Start2	Start3	Start4	Start5	Start6	Start7	Start8	Start9	Start10	Start11	Start12	T	_TYPE_	_RHS_
1	-1	1	.	.	.	.	.	.	.	.	.	.	.	.	GE	0
2	-1	.	1	.	.	.	.	.	.	.	.	.	.	.	GE	0
3	-1	.	.	1	.	.	.	.	.	.	.	.	.	.	GE	0
4	-1	.	.	.	.	.	1	.	.	.	.	.	.	.	GE	0
5	-1	.	.	.	.	.	.	1	.	.	.	.	.	.	GE	0
6	.	-1	.	.	1	.	.	.	.	.	.	.	.	.	GE	1
7	.	.	-1	.	.	1	.	.	.	.	.	.	.	.	GE	2
8	.	.	.	-1	.	.	.	.	.	.	.	.	.	1	GE	2
9	.	.	.	.	-1	.	.	.	.	.	.	.	.	1	GE	6
10	.	.	.	.	.	-1	.	.	.	.	.	.	.	1	GE	3
11	.	.	.	.	.	.	-1	.	.	.	.	.	1	.	GE	6
12	.	.	.	.	.	.	.	-1	1	.	.	.	.	.	GE	1
13	.	.	.	.	.	.	.	-1	.	1	.	.	.	.	GE	1
14	.	.	.	.	.	.	.	-1	.	.	1	.	.	.	GE	1
15	.	.	.	.	.	.	.	.	-1	.	.	1	.	.	GE	3
16	.	.	.	.	.	.	.	.	.	-1	.	.	1	.	GE	3
17	.	.	.	.	.	.	.	.	.	.	-1	.	1	.	GE	4
18	.	.	.	.	.	.	.	.	.	.	.	-1	1	.	GE	2
19	.	.	.	.	.	.	.	.	.	.	.	.	-1	1	GE	4

Finally, the Objective data set shown in Output 3.15.3 specifies the objective function of this problem.

Output 3.15.3 Objective Function

Obs	T	_TYPE_	_RHS_
1	1	MIN	.

The following statements merge these three data sets to create the Constraint data set that the CLP procedure uses to find a schedule that minimizes the makespan.

```
data ConData;
   set StartTime Precedence Objective;
run;
```

```
proc clp condata=ConData out=OutData usecondatavars=1;
   /* set lower and upper bounds for the objective function */
   obj lb=10 ub=40;

   /* Post a cumulative constraint for the resource */
   cumulative (start=(Start1-Start14)
               duration=(1 2 2 6 3 6 1 3 3 4 2 4 1 1)
               demand=(4 2 3 3 2 3 1 2 2 1 2 2 4 8)
               capacity=8);
run;
```

The USECONDATAVARS=1 specification in the PROC CLP statement indicates that all variables are implicitly defined in the Constraint data table ConData. The OBJ statement specifies the lower and upper bounds on the objective function. In this example, they are actually the lower and upper bounds of variable T, the completion time of the project. The CUMULATIVE statement defines the task start times, task durations, and task demands, and it enforces the constraint that the cumulative resource usage of Task 1 though Task 14 not exceed the resource capacity at any point in time.

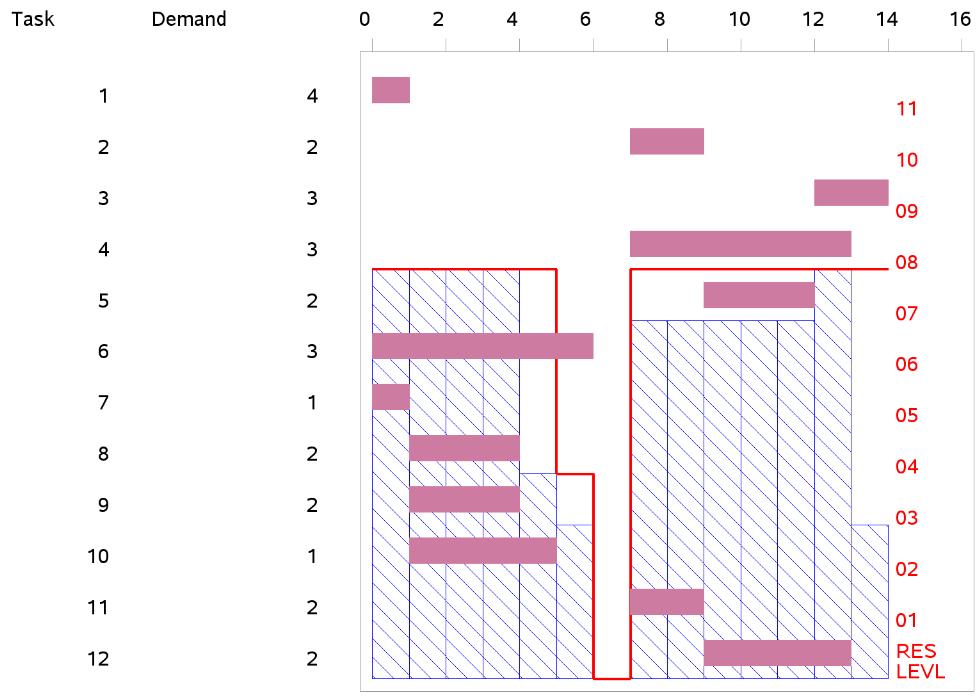
The table in Output 3.15.4, which is derived from the solution table, shows a corresponding schedule for the 12 tasks.

Output 3.15.4 RCPSP/max Schedule with Calendars

Task	Start	Duration	End	Demand
1	0	1	1	4
2	7	2	9	2
3	12	2	14	3
4	7	6	13	3
5	9	3	12	2
6	0	6	6	3
7	0	1	1	1
8	1	3	4	2
9	1	3	4	2
10	1	4	5	1
11	7	2	9	2
12	9	4	13	2

The Gantt chart shown in Output 3.15.5 reports the resource demand and displays the resource-constrained schedule for each of the 12 tasks. The chart also overlays a plot of the resource capacity with time and a histogram of the resource demand with time. The resource quantity axis is on the right. Notice the decreased consumption in time periods 5 and 6.

Output 3.15.5 Gantt Chart Showing Task Schedule and Resource Consumption



Scheduling with Optional Tasks

The second part of this example illustrates how you can use the cumulative constraint to model optional tasks. Optional tasks are tasks that might or might not occur; they can arise in scenarios such as project selection, job contracting, and so on.

Consider the previous resource-constrained project scheduling problem with calendars that completed in 14 time periods. Suppose that the project must now be completed in 12 time periods and can assign certain tasks to external contractors at additional cost in order to complete in fewer time periods. Assume that Task 1, Task 2, Task 3, Task 4, and Task 5 can be contracted out subject to the following pairwise conditions:

- If Task 1 is contracted out, then Task 4 also be contracted out, and vice-versa. The cost of jointly contracting out Task 1 and Task 4 is $15,000.
- Similarly, Task 2 and Task 5 must be either jointly contracted out or processed in-house together. The cost of jointly contracting out Task 2 and Task 5 is $12,000.
- The cost of contracting out Task 3 is $5,000.

The objective is to minimize the total cost, subject to finishing the project within 12 time periods.

The StartTime*2* data set shown in Output 3.15.6 specifies the new lower and upper bounds of the start-time variables for all tasks. Note that the variable T is now fixed to 12 to reflect the new makespan restriction. Although not necessary, the upper bounds of all variables Start1 to Start12 can also be set to 12.

Output 3.15.6 StartTime2 Data Set

Obs	Start1	Start2	Start3	Start4	Start5	Start6	Start7	Start8	Start9	Start10	Start11	Start12
1	0	0	0	0	0	0	0	0	0	0	0	0
2	12	12	12	12	12	12	12	12	12	12	12	12
3	.	.	.	.	.	.	.	.	.	.	.	.

Obs	Start13	Start14	S	T	_TYPE_	_RHS_
1	.	.	.	.	LOWERBD	.
2	.	.	.	.	UPPERBD	.
3	5	6	0	12	FIXED	.

One way to model the fact that tasks might or might not be executed in-house is to introduce a demand variable for each such task. The lower bound for this demand variable is equal to 0, and its upper bound is equal to the demand it would have had if it had been processed in-house. The demand variable for a task is assigned to its lower bound if the task is contracted out, or to its upper bound if the task is not contracted out. The usage variables are declared in the Demand data set as shown in Output 3.15.7.

Output 3.15.7 Demand Data Set

Obs	Demand1	Demand2	Demand3	Demand4	Demand5	Demand6	Demand7	Demand8	Demand9
1	0	0	0	0	0	.	.	.	.
2	4	2	3	3	2	.	.	.	.
3	.	.	.	.	.	2	1	1	2

Obs	Demand10	Demand11	Demand12	Demand13	Demand14	_TYPE_	_RHS_
1	.	.	.	.	.	LOWERBD	.
2	.	.	.	.	.	UPPERBD	.
3	1	2	2	4	8	FIXED	.

A binary variable is created for each task that can be contracted out. The value of the binary variable is 1 if the corresponding task is contracted out, or 0 if is not contracted out. The relationship between the binary variable X_i and demand variable Demand_i for task i can be expressed by the linear constraint

$$\mathrm{Demand}_i = \mathrm{demand}[i] \times (1 - \mathrm{X}_i)$$

where $\mathrm{demand}[i]$ is the demand for the task i.

Because Tasks 1 and 4 and Tasks 2 and 5 must be jointly processed, the number of binary variables can be reduced to three.

$$X_1 = \begin{cases} 1 & \text{if Task 1 and Task 4 are contracted out} \\ 0 & \text{otherwise} \end{cases}$$

$$X_2 = \begin{cases} 1 & \text{if Task 2 and Task 5 are contracted out} \\ 0 & \text{otherwise} \end{cases}$$

$$X_3 = \begin{cases} 1 & \text{if Task 3 is contracted out} \\ 0 & \text{otherwise} \end{cases}$$

The Binary data set shown in Output 3.15.8 declares these three binary variables.

Output 3.15.8 Binary Variables

Obs	X1	X2	X3	_TYPE_	_RHS_
1	1	1	1	BINARY	.

The Usage data set shown in Output 3.15.9 contains the linear constraints that relate the binary variables and the demand variables.

Output 3.15.9 Usage Constraints

Obs	Demand1	Demand2	Demand3	Demand4	Demand5	X1	X2	X3	_TYPE_	_RHS_
1	1	.	.	.	.	4	.	.	EQ	4
2	.	1	.	.	.	.	2	.	EQ	2
3	.	.	1	.	.	.	.	3	EQ	3
4	.	.	.	1	.	3	.	.	EQ	3
5	.	.	.	.	1	.	2	.	EQ	2

Finally, the objective function shown in Output 3.15.10 defines the total contracting cost of the project that is to be minimized.

Output 3.15.10 Objective Function

Obs	X1	X2	X3	_TYPE_	_RHS_
1	15	12	5	MIN	.

As before, the data sets are merged to create the Constraint data table, and the CLP procedure is invoked to find the best solution as follows:

```
data ConData2;
   set StartTime2 Precedence Demand Binary Usage Objective2;
run;


proc clp condata=ConData2 out=OutData2 usecondatavars=1;
   /* set lower and upper bounds for the objective function */
   obj lb=0 ub=32;

   /* Define a cumulative constraint for the resource */
   cumulative (start=(Start1-Start14)
               demand=(Demand1-Demand14)
               dur=(1 2 2 6 3 6 1 3 3 4 2 4 1 1)
               capacity=8);
run;
```

The table in Output 3.15.11, derived from the Solution data set, displays the optimal schedule that has a makespan of 11 time periods and contracts out Task 1 and Task 4.

Output 3.15.11 RCPSP Schedule with Optional Tasks

Task	Start	Duration	End	Demand
2	0	2	2	2
3	7	2	9	3
5	2	3	5	2
6	0	6	6	2
7	0	1	1	1
8	1	3	4	1
9	1	3	4	2
10	1	4	5	1
11	4	2	6	2
12	7	4	11	2

Example 3.16: Resource-Constrained Scheduling Problem with Nonstandard Temporal Constraints

This example illustrates the use of the cumulative constraint to solve a resource-constrained project scheduling problem that has nonstandard temporal constraints.

For more information about this example, see the same example in "Example 3.9: Resource-Constrained Scheduling with Nonstandard Temporal Constraints" which illustrates the capabilities of the scheduling solver.

The following constraints are added to the standard precedence constraints:

- The time between the completion of a particular formwork and the completion of its corresponding concrete foundation is at most four days:

  $$f(\mathrm{s}i) \geq f(\mathrm{b}i) - 4, \quad i = 1, \ldots, 6$$

- There are at most three days between the end of a particular excavation (or foundation piles) and the beginning of the corresponding formwork:

  $$f(\mathrm{a}i) \geq s(\mathrm{s}i) - 3, \quad i = 1, 2, 5, 6$$

  and

  $$f(\mathrm{p1}) \geq s(\mathrm{s3}) - 3$$

  $$f(\mathrm{p2}) \geq s(\mathrm{s4}) - 3$$

- The formworks must start at least six days after the beginning of the erection of the temporary housing:

  $$s(\mathrm{s}i) \geq s(\mathrm{ue}) + 6, \quad i = 1, \ldots, 6$$

- The removal of the temporary housing can start at most two days before the end of the last masonry work:

$$s(\mathrm{ua}) \geq f(\mathrm{m}i) - 2, \quad i = 1, \ldots, 6$$

- The delivery of the preformed bearers occurs exactly 30 days after the beginning of the project:

$$s(\mathrm{l}) = s(\mathrm{pa}) + 30$$

For convenience, the activity name is used to represent the start time variable of the activity. Each temporal constraint is represented by a linear constraint in the Constraint data set. An additional observation is added to the Constraint data set to define the objective of minimizing the makespan, which in this case is the milestone pe, indicating the end of the project. The CLP procedure is invoked with the WDEG variable selection strategy along with seven cumulative constraints, one corresponding to each of the resources: Excavator, Pile Driver, Carpentry, Concrete Mixer, Bricklaying, Crane, and Caterpillar.

```
proc clp condata=condata varselect=wdeg out=out usecondatavars=1;
   obj lb=0 ub=2000;
   cumulative (start=( m1-m6 ) dur=( 16 8 8 8 8 20 ))
              (start=( s1-s6 ) dur=( 8 4 4 4 4 10 ))
              (start=( v1 v2 ) dur=( 15 10 ))
              (start=( b1-b6 ) dur=( 1 1 1 1 1 1))
              (start=( l t1-t5 ) dur=( 2 12 12 12 12 12 ))
              (start=( a1-a6 ) dur=( 4 2 2 2 2 5))
              (start=( p1 p2 ) dur=( 20 13 ));
run;
```

The solution data set shown in Output 3.16.1 displays the corresponding start times for all the tasks.

Output 3.16.1 Bridge Construction Task Start Times

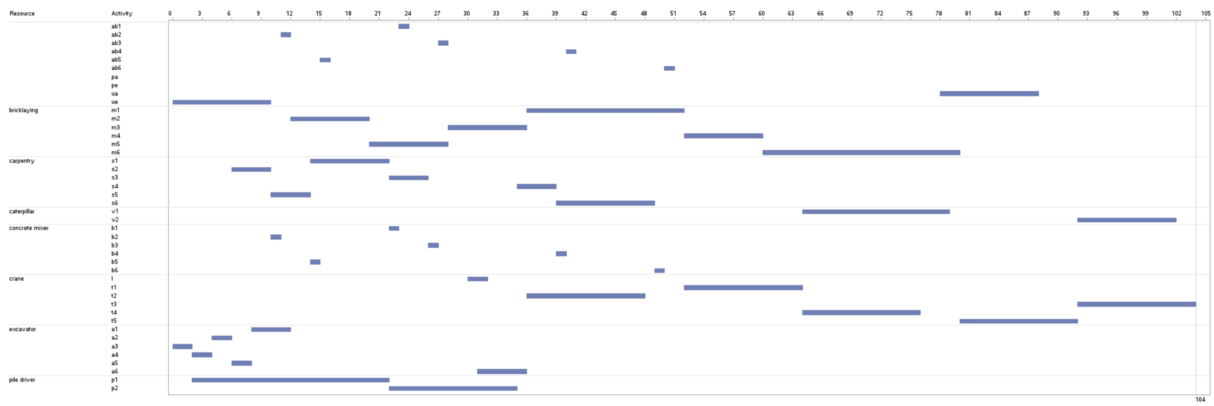
s1	a1	s2	a2	p1	a3	p2	a4	s5	a5	s6	a6	m1	ab1	m2	ab2	m3	ab3	m4	ab4	m5	ab5	m6	ab6	b1	b2
14	8	6	4	2	0	22	2	10	6	39	31	36	23	12	11	28	27	52	40	20	15	60	50	22	10

b3	s3	b4	s4	b5	b6	t1	l	t2	t3	t4	t5	ua	pa	ue	v1	pe	v2
26	22	39	35	14	49	52	30	36	92	64	80	78	0	0	64	104	92

A Gantt chart of the resulting schedule, zoned by resource, is shown in Output 3.16.2.

Output 3.16.2 Bridge Construction Schedule

Zoned by Resource



Statement and Option Cross-Reference Table

Table 3.15 shows which examples in this section use each of the statements and options in the CLP procedure.

Table 3.15 Statements and Options Specified in Examples 3.1–3.14

Statement	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
ACTIVITY										X						
ALLDIFF	X				X							X		X		
CUMULATIVE															X	X
ELEMENT			X	X	X		X									
GCC		X	X			X	X						X			
LEXICO													X			
LINCON	X	X	X	X	X	X	X					X		X		
OBJ						X									X	X
PACK														X		
REIFY					X							X	X	X		

Table 3.15 (continued)

Statement	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
REQUIRES								X		X						
RESOURCE								X		X						
SCHEDULE								X	X	X	X					
VARIABLE	X	X	X	X	X	X	X					X	X	X		
Option	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
ACTDATA=								X	X		X					
ACTSELECT=								X		X						
CONDATA=				X		X									X	X
DOMAIN=										X	X		X			
DPR=										X	X					
DURATION=								X		X	X					
EDGEFINDER=										X	X					
EVALVARSEL=	X															
FINDALLSOLNS					X		X					X				
FINISH=									X							
LB=						X									X	X
MAXTIME=	X															
NOTFIRST=											X					
NOTLAST=											X					
OUT=	X	X	X	X	X	X	X					X	X	X	X	X
RESTARTS=										X	X					
SCHEDRES=										X						
SCHEDTIME=									X	X						
SCHEDULE=								X			X					
SHOWPROGRESS										X	X					
START=										X						
UB=						X									X	X
USECONDATAVARS=															X	X
VARASSIGN=													X			
VARSELECT=	X					X	X					X	X	X		X

References

Applegate, D. L., and Cook, W. (1991). “A Computational Study of the Job Shop Scheduling Problem.” *ORSA Journal on Computing* 3:149–156.

Baptiste, P., and Le Pape, C. (1996). “Edge-Finding Constraint Propagation Algorithms for Disjunctive and Cumulative Scheduling.” In *Proceedings of the Fifteenth Workshop of the UK Planning Special Interest Group (PLANSIG)*. Liverpool.

Bartusch, M. (1983). “Optimierung von Netzplänen mit Anordnungsbeziehungen bei knappen Betriebsmitteln.” Ph.D. thesis, Universität Passau, Fakultät für Mathematik und Informatik.

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.

Carlier, J., and Pinson, E. (1989). "An Algorithm for Solving the Job-Shop Scheduling Problem." *Management Science* 35:164–176.

Carlier, J., and Pinson, E. (1990). "A Practical Use of Jackson's Preemptive Schedule for Solving the Job-Shop Problem." *Annals of Operations Research* 26:269–287.

Colmerauer, A. (1990). "An Introduction to PROLOG III." *Communications of the ACM* 33:70–90.

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., Pearson, J., 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. Berlin: 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.

Henz, M. (2001). "Scheduling a Major College Basketball Conference—Revisited." *Operations Research* 49:163–168.

Jaffar, J., and Lassez, J. (1987). "Constraint Logic Programming." In *Proceedings of the Fourteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 111–119. New York: Association for Computing Machinery.

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

Lawrence, S. (1984). *Resource Constrained Project Scheduling: An Experimental Investigation of Heuristic Scheduling Techniques (Supplement)*. Technical report, Graduate School of Industrial Administration, Carnegie Mellon University.

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.

Muth, J. F., and Thompson, G. L., eds. (1963). *Industrial Scheduling*. Englewood Cliffs, NJ: Prentice-Hall.

Nemhauser, G. L., and Trick, M. A. (1998). "Scheduling a Major College Basketball Conference." *Operations Research* 46:1–8.

Nemhauser, G. L., and Wolsey, L. A. (1988). *Integer and Combinatorial Optimization*. New York: John Wiley & Sons.

Nuijten, W. (1994). "Time and Resource Constrained Scheduling." Ph.D. diss., Eindhoven Institute of Technology, Netherlands.

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." `https://mat.tepper.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.

Van Hentenryck, P., Deville, Y., and Teng, C. (1992). "A Generic Arc-Consistency Algorithm and Its Specializations." *Artificial Intelligence* 57:291–321.

Vilím, P. (2011). "Timetable Edge Finding Filtering Algorithm for Discrete Cumulative Resources." In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR 2011)*, edited by T. Achterberg and J. C. Beck, vol. 6697 of Lecture Notes in Computer Science, 230–245. Heidelberg: Springer.

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.

Williams, H. P., and Wilson, J. M. (1998). "Connections between Integer Linear Programming and Constraint Logic Programming: An Overview and Introduction to the Cluster of Articles." *INFORMS Journal on Computing* 10:261–264.

Subject Index

Syntax Index