Chapter 1

The CLP Procedure (Experimental)

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 solver 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.

For the most recent updates to the documentation for this experimental procedure, see the Statistics and Operations Research Documentation page at http://support.sas.com/rnd/app/doc.html.

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 taking values from finite domains and a finite set of constraints restricting the values 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* restricting the values that the variables can simultaneously take.

Note that 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 as well as the resource(s) that must be assigned to the activity corresponding to the start time.

A solution to a CSP is an assignment of values to the variables in order to satisfy all the constraints, and the problem amounts to finding solution(s), 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 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 due to the late detection of conflicts; that is, it is oriented toward *recovering* from failures and not *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 is encountering repeated failures due to the same reason, sometimes referred to as "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 employs an active use, rather than a passive use, of constraints.

Constraint Propagation

A more efficient technique than backtracking is that of 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 referred to as 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 10x10 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 the domain pruning scheme that is followed—domain propagation or interval propagation. 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 on and that the current set of alternatives 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 Elliot 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 two types: standard CSPs and scheduling CSPs. 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. 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. The CSP type is determined by the presence of either the OUT= option or the SCHEDDATA= option in the PROC CLP statement.

Specifying the OUT= option in the PROC CLP statement indicates to the CLP procedure that the CSP is a standard type. As such, the procedure will expect VAR, LINCON, REIFY, ALLDIFF, ARRAY, and FOREACH statements. You can also specify a Constraint data set by using the CONDATA= option in the PROC CLP statement in lieu of, or in combination with, VAR and LINCON statements.

Specifying the SCHEDDATA= option in the PROC CLP statement indicates to the CLP procedure that the CSP is a scheduling type. As such, the procedure will expect ACTIVITY, RESOURCE, REQUIRES, and SCHEDULE statements. You can also specify an Activity data set by using the ACTDATA= option in the PROC CLP statement in lieu of, or in combination with, the ACTIVITY statement. Precedence relationships between activities must be defined using the ACTDATA= data set. Resource requirements of activities must be defined using the RESOURCE and REQUIRES statements.

The output data set contains any solutions determined by the CLP procedure. For more information about the format and layout, see the section "Details: CLP Procedure" on page 24.

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

Once 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.

Introductory Examples: CLP Procedure

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

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:

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

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 VAR statement identifies the variables to 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 is used to define the linear constraint SEND + MORE = MONEY, as well as 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 VAR 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 output data set  */
   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 produced by the CLP procedure is shown in Figure 1.1.

Figure 1.1 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 determined by the CLP procedure is as follows:

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

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, you should have the following for all i and j:

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

and

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

In other words, you should have

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

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, including the one 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 VAR statement to be the digits 1 through 8 since they represent the row number on an 8x8 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 they are encountered by the CLP procedure.

The following statements display the solution data set shown in Figure 1.2:

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

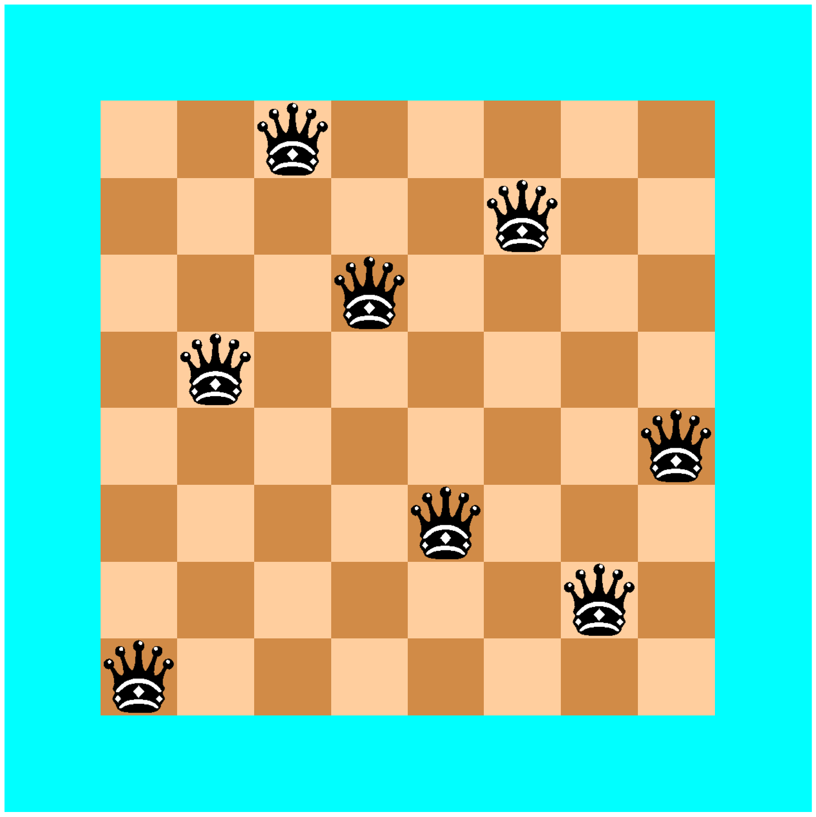
Figure 1.2 A Solution to the Eight Queens Problem

```
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 1.3.

Figure 1.3 A Solution to the Eight Queens Problem



Syntax: CLP Procedure

The following statements are used in PROC CLP:

PROC CLP *option(s)* ;
    ACTIVITY *specification* < . . . > ;
    ALLDIFF *(variables)* < . . . > ;
    ARRAY *specification* < . . . > ;
    FOREACH *(array, type, < offset>)* ;
    LINCON *linear_constraint* < , . . . > ;
    REIFY *variable : (linear_constraint)* < . . . > ;
    REQUIRES *specification* < . . . > ;
    RESOURCE *specification* < . . . > ;
    SCHEDULE *option(s)* ;
    VAR *specification* < . . . > ;

Functional Summary

The statements and options available with PROC CLP are summarized by purpose in Table 1.1.

Table 1.1 Functional Summary

Description	Statement	Option
Assignment Strategy Options		
variable assignment strategy	PROC CLP	VARASSIGN=
activity assignment strategy	SCHEDULE	ACTASSIGN=
Data Set Options		
activity input data set	PROC CLP	ACTDATA=
constraint input data set	PROC CLP	CONDATA=
solution output data set	PROC CLP	OUT=
schedule output data set	PROC CLP	SCHEDDATA=
General Options		
suppress preprocessing	PROC CLP	NOPREPROCESS
upper bound on CPU time (seconds)	PROC CLP	MAXTIME=
Output Control Options		
find all possible solutions	PROC CLP	FINDALLSOLNS
indicate progress in log	PROC CLP	SHOWPROGRESS
number of solutions	PROC CLP	SOLNS=
Scheduling CSP-Related Statements		
activity specifications	ACTIVITY	
resource requirement specifications	REQUIRES	

Table 1.1 *continued*

Description	Statement	Option
resource specifications	RESOURCE	
scheduling parameters	SCHEDULE	
Scheduling: Resource Constraints		
edge-finder consistency routines	SCHEDULE	EDGEFINDER=
not first edge-finder extension	SCHEDULE	NOTFIRST=
not last edge-finder extension	SCHEDULE	NOTLAST=
Scheduling: Temporal Constraints		
activity duration	ACTIVITY	DURATION=
activity finish lower bound	ACTIVITY	FGE=
activity finish upper bound	ACTIVITY	FLE=
activity start lower bound	ACTIVITY	SGE=
activity start upper bound	ACTIVITY	SLE=
schedule duration	SCHEDULE	DURATION=
schedule finish	SCHEDULE	FINISH=
schedule start	SCHEDULE	START=
Scheduling: Search Control Options		
dead-end multiplier	PROC CLP	DM=
number of allowable dead ends per restart	PROC CLP	DPR=
number of search restarts	PROC CLP	RESTARTS=
Selection Strategy Options		
variable selection strategy	PROC CLP	VARSELECT=
activity selection strategy	SCHEDULE	ACTSELECT=
Standard CSP Statements		
all-different constraints	ALLDIFF	
array specifications	ARRAY	
for-each constraints	FOREACH	
linear constraints	LINCON	
reified constraints	REIFY	
variable specifications	VAR	

PROC CLP Statement

PROC CLP *option(s)* **;**

The following options can appear in the PROC CLP statement.

ACTDATA=*SAS-data-set*

ACTIVITY=*SAS-data-set*

identifies the input 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. The activities and time alignment constraints can also be directly specified 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 linear constraints, variable types, and variable bounds. 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. The linear constraints can also be specified in-line using the LINCON statement. The CLP procedure enables you to define linear constraints by using a combination of the two specifications. When defining linear constraints, you must define the structural variables by using a VAR statement. Note that variable bounds can be defined using the VAR statement, and any such definitions override those defined in the CONDATA= data set.

DM=*m*

DEM=*m*

specifies the dead-end multiplier for the 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 and the number of unassigned activities. The default value is 0.15. This option is valid only with the SCHEDDATA= 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 VAR statement or the CONDATA= data set. The default is $[0,\infty]$.

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. The default value of the DPR= option is ∞.

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 scheduling-related problems. Therefore, this option is ignored when you are solving a scheduling-related problem.

MAXTIME=*m*

specifies an upper bound on the number of CPU seconds allocated for solving the problem. Note that 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. 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 the solution(s) to the CSP, if any exist(s). Each observation in the OUT= data set corresponds to a solution of the CSP. The number of solutions generated can be controlled using the SOLNS= option in the PROC CLP statement.

RESTARTS=*n*

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

SCHEDDATA=*SAS-data-set*

SCHEDULE=*SAS-data-set*

identifies the output data set that contains the scheduling-related solution to the CSP, if one exists. Each observation in the SCHEDDATA= data set corresponds to an activity. The format of the schedule data set is similar to the schedule data set generated by the CPM and PM procedures in SAS/OR software. The number of solutions generated can be controlled using the SOLNS= 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.

SOLNS=*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 SOLNS= parameter.

VARASSIGN=*keyword*

specifies the value selection strategy. Currently there is only one value selection strategy. The MIN strategy selects the minimum value from the domain of the selected variable. To assign activities, use the ACTASSIGN= option in the SCHEDULE statement.

VARSELECT=*keyword*

specifies the variable selection strategy. Both static and dynamic strategies are available. Possible values follow.

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

Dynamic strategies are as follows:

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

The dynamic strategies embody the "Fail First Principle" (FFP) of Haralick and Elliot (1980), which suggests that "To succeed, try first where you are most likely to fail." The default strategy is MINR. To select activities, use the ACTSELECT= option in the SCHEDULE statement.

ACTIVITY Statement

ACTIVITY *specification < . . . >* **;**

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

activity < = (< DUR= > duration < type=date . . . >) >

(activity_list) < = (< DUR= > duration < type=date . . . >) >

where *duration* is the activity duration and *type* is a keyword specifying an alignment-type constraint on the activity (or activities) with respect to the date given by *date*.

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 default duration is 0.

Valid *type* keywords are as follows:

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

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, you would specify the following:

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

You can alternatively use the ACTDATA= data set to define the 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. The SCHEDDATA= option must be specified when the ACTIVITY statement is used.

ALLDIFF Statement

ALLDIFF *(variables)* < . . . > **;**

ALLDIFFERENT *(variables)* < . . . > **;**

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 declared in the VAR statement.

ARRAY Statement

ARRAY *specification* < . . . > **;**

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 VAR statement. The ARRAY statement is required when you are specifying a constraint by FOREACH statement.

FOREACH Statement

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

where *array* must be defined using an ARRAY statement, *type* is a keyword that determines the type of the constraint, and *offset* is an integer. The default value is 0.

The FOREACH statement iteratively applies a constraint over an array of variables. The type of the constraint is determined by *type*. The optional *offset* parameter is an integer and is interpreted in the context of the constraint type.

Currently, the only valid *type* keyword is **DIFF**.

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

$$\text{variable\_}i + \text{offset} \times i \;\neq\; \text{variable\_}j + \text{offset} \times j \qquad \forall\, i \neq j,\; i, j = 1, \ldots, \text{array\_dimension}$$

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

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

LINCON Statement

LINCON *linear_constraint* < , . . . > **;**

LINEAR *linear_constraint* < , . . . > **;**

A *linear_constraint* is specified in the following form:

linear_term_1 operator linear_term_2

where a *linear_term* is of the form

((<+|-> *variable* | *number* <* *variable* >). . .)

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

<, <=, =, >=, >, <>, LE, EQ, GE, LT, 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 $4x_1 - 3x_2 = 5$ is expressed as

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

and the constraints

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

are 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 using the CONDATA= data set. Regardless of the specification, you must define the variables by using a VAR statement.

REIFY Statement

REIFY *variable : (linear_constraint)* < . . . > **;**

A *linear_constraint* is specified in the following form:

linear_term_1 operator linear_term_2

where a *linear_term* is of the form

((<+|-> *variable* **|** *number* **<*** *variable* **>)**. . . **)**

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

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

The REIFY statement associates a binary variable with a linear constraint. The value of the binary variable is 1 or 0 depending on whether the linear constraint is satisfied or not, respectively. The linear constraint is said to be reified, and the binary variable is referred to as the control variable. As with the other variables, the control variable must also be defined in a VAR 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 REIFY constraint can also be used to express a constraint involving 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 *specification* < . . . > **;**

REQ *specification* < . . . > **;**

A REQUIRES *specification* is in the following form:

activity = *(resource* < , . . . >*)*

where *activity* represents a single activity or a list of activities. Likewise *resource* represents a single resource or a list of resources.

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. The order of appearance of the ACTIVITY and REQUIRES statements and ACTIVITY dataset affect the DET strategy. For example, to specify that activity A requires resource R, you would need the following statements:

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

Sometimes, 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 can be defined by multiple *resource* specifications separated by commas. For example, to specify that the requirements of activity A could be satisfied by either R1, R2, or R3, you would need the following statements:

```
activity A;
resource R1 R2 R3;
requires A = (R1, R2, R3);
```

It is also possible that an activity might require more than one resource simultaneously. The specification is similar, the only difference being that the simultaneous requirement is specified without any commas separating them.

For example, the following statements specify that activity A and B requires resources R1 and R2 simultaneously *or* resources R3 and R4 simultaneously:

```
activity A B;
resource (R1-R4);
requires (A B) = ((R1 R2), (R3 R4));
```

RESOURCE Statement

RESOURCE *specification* < . . . > **;**

RES *specification* < . . . > **;**

A RESOURCE *specification* is a single resource or a list of resources.

The RESOURCE statement specifies the names of all resources that are available to be allocated to the activities. The REQUIRES statement is necessary to specify the resource requirements of an activity. Currently all resources are assumed to be unary resources in that their capacity is equal to one and they cannot be assigned to more than one activity at any given time.

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 setting of the ACTSELECT= option. After an activity has been selected, the activity assignment strategy determines a start time and a set of resources (empty if the activity has no resource requirements) for the selected activity. The interpretation of the assignment strategy depends on whether the *activity selection* strategy has been specified as RJRAND or not.

The activity is assigned its earliest possible start time unless the activity selection strategy (ACTSELECT=) is RJRAND; otherwise the activity is assigned its latest possible start time.

Figure 1.4 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 supported by each of the resources for the duration of the activity.

Figure 1.4 Potential Activity Start Times

For example, if ACTSELECT=LJRAND, the activity is assigned a start time of 6 and one of R1 or R2 is assigned. On the other hand, if ACTSELECT=RJRAND, the activity is assigned a start time of 13 and one of R4, R5, or R6 is assigned.

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

RAND	randomly selects a set of resources that support the selected start time for the activity. In Figure 1.4, if the activity start time is set to 6, the strategy randomly selects between R1 and R2. Otherwise, the strategy randomly selects among R4, R5, and R6.
MAXTW | MAXLS	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. In Figure 1.4, if the activity start time is set to 6, the resources that support the selected start time are R1 and R2. Since R1 has a smaller time window, the strategy selects R2. On the other hand, if the activity

start time is set to 13, the resources that support the selected start time are R4, R5, and R6. Because R4 has a smaller time window than R5 or R6, the strategy randomly selects between R5 and R6.

The default 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 are selection strategies that use a random heuristic to break ties.

LJRAND | RAND	selects an activity at random from those that begin prior to the earliest early finish time. This strategy was proposed by Nuijten (1994).
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 date.
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 date.
DMINLS	selects the activity with the earliest late start time.

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

1. ACTIVITY statement
2. REQUIRES statement
3. ACTIVITY dataset

The default 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 problems. 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 requiring a given resource or set of resources and prunes the domain of activity as appropriate.

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 requiring a given resource or set of resources and prunes its domain as appropriate.
- LAST: The edge-finder algorithm attempts to determine whether an activity must be processed last from a set of activities requiring a given resource or set of resources and prunes its domain as appropriate.
- 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 related to edge-finder consistency routines, see "Details: CLP Procedure" on page 24.

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 problems. 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 requiring a given resource or set of resources and prunes its domain as appropriate.

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. It should be noted that more propagation usually comes at a higher cost—mainly that of performance. The challenge is to strike the right balance. Specifying the NOTFIRST= option implicitly turns on the EDGEFINDER=LAST option since the latter is a special case of the former.

There is a corresponding option NOTLAST=, which determines whether an activity cannot be the last to be processed from a set of activities requiring a given resource or set of resources.

For more information about options related to edge-finder consistency routines, see the section "Details: CLP Procedure" on page 24.

NOTLAST=*level*

NL=*level*

activates an extension of the edge-finder consistency routines for scheduling problems. 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 requiring a given resource or set of resources and prunes its domain as appropriate.

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. It should be noted that more propagation usually comes at a higher cost—mainly that of performance. The challenge is to strike the right balance. Specifying the NOTLAST= option implicitly turns on the EDGEFINDER=FIRST option since the latter is a special case of the former.

There is a corresponding option NOTFIRST=, which 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 related to edge-finder consistency routines, see the section "Details: CLP Procedure" on page 24.

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.

VAR Statement

VAR *specification* < . . . > **;**

A VAR specification can be one of the following types:

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

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

The VAR statement declares all the variables that are to be considered in the CSP and, optionally, defines their domains. Any variable domains defined in a VAR statement override the global variable domains defined using the DOMAIN= option in the PROC CLP statement as well as any bounds defined using the CONDATA= data set. If *lower-bound* is specified and *upper-bound* is omitted, the corresponding variables are considered as being assigned to *lower-bound*.

Details: CLP Procedure

This section provides a detailed outline of the use of the CLP procedure. The material is organized in subsections that describe different aspects of the procedure.

Modes of Operation

The CLP procedure can be invoked in one of two modes: standard mode and scheduling mode. The standard mode gives you access to linear constraints, reified constraints, all-different constraints, and array constraints; the scheduling mode gives you access to more scheduling-specific constraints, such as temporal constraints (precedence and time) and resource constraints. In standard mode, the decision variables are one-dimensional; a variable is assigned an integer in a solution. In scheduling mode, the variables are typically multidimensional; a variable is assigned a start time and possibly a set of resources in a solution. In scheduling mode, the variables are referred to as activities, and the solution is referred to as a schedule.

Selecting the Mode of Operation

The CLP procedure requires the specification of an output data set to store the solution(s) to the CSP. There are two possible output data sets: the Solution data set (specified using the OUT= option in the PROC CLP statement), which corresponds to the standard mode of operation, and the Schedule data set (specified using the SCHEDDATA= option in the PROC CLP statement), which corresponds to the scheduling mode of operation. The mode is determined by which output data set has been specified. If an output data set is not specified, the procedure terminates with an error message. If both output data sets have been specified, the Schedule data set is ignored.

Constraint Data Set

The Constraint data set defines linear constraints, variable types, and bounds on variable domains. You can use a Constraint data set in lieu of, or in combination with, a LINCON and/or a VAR statement in order to define linear constraints, variable types, and variable bounds. The Constraint data set is similar to the problem data set input to the LP procedure in SAS/OR software and is specified 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, the _RHS_ variable, and at least one numeric variable. In the absence of this requirement, 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, and FIXED. 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, 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 VAR statement is interpreted as a structural variable for the linear constraint.

EQ (=) defines a linear equality

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

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 disequation 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 values LOWERBD and UPPERBD specify additional lower bounds and upper bounds on the variable domains, respectively. In an observation where the _TYPE_ variable is equal to LOWERBD, a nonmissing value for a decision variable is considered a lower bound for that variable. Similarly, in an observation where the _TYPE_ variable is equal to UPPERBD, a nonmissing value for a decision variable is considered an upper bound for that variable. Note that lower and upper bounds as previously defined will be overridden by lower and upper bounds defined using a VAR statement.

Variable Types

The keywords BINARY and FIXED are interpreted as specifying numeric types. If the value of _TYPE_ is BINARY for an observation, then any decision variable with a nonmissing 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 nonmissing entry for the observation is interpreted as being assigned to that nonmissing 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 $\{c\}$. It is important to note that the value c should belong to the domain of X, or the problem is deemed infeasible.

Variables in the CONDATA= Data Set

Table 1.2 lists all the variables associated with the Constraint data set and their interpretations by the CLP procedure. The table also lists for each variable its type (C for character, N for numeric), the possible values it can assume, and its default value.

Table 1.2 Constraint Data Set Variables

Name	Type	Description	Allowed Values	Default
TYPE	C	observation type	EQ, LE, GE, NE, LT, GT, LOWERBD, UPPERBD, BINARY, FIXED	
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 SOLNS= option in the PROC CLP statement. If you prefer that CLP determine all possible solutions instead, you can specify the FINDALLSOLNS option in the PROC CLP statement.

The solution data set contains as many decision variables as have been defined in the call to PROC CLP. 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 relating to the activities. The activity data set is similar to the activity data set input to the CPM procedure in SAS/OR software and is specified using the ACTDATA= option in the PROC CLP statement.

The activity data set enables you to define an activity, its domain, and any temporal constraints. The temporal constraints could be either time-alignment-type or precedence-type constraints. The activity data set requires, at the minimum, 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, and the duration is determined with the _DURATION_ variable. In addition to the mandatory variables, you can also specify temporal constraints related to the activities.

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 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. If the _ALIGNDATE_ is present but the _ALIGNTYPE_ variable is missing, the alignment type is assumed to be SGE. The _ALIGNTYPE_ variable can take the values shown in Table 1.3.

Table 1.3 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 using AON (activity-on-node) format. The _SUCCESSOR_ variable must have the same type as that of the _ACTIVITY_ 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 given by *lagdur*. The interpretation of each of the different lag types is given in Table 1.4.

Table 1.4 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/finish times of B, while the last four types force the start/finish times to be set equal to the lower bound of the domain. This enables 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 activity data set cannot be used to define resource-requirement-type constraints. To define these constraints, you must specify RESOURCE and REQUIRES statements.

Variables in the ACTDATA= data set

Table 1.5 lists all the variables associated with the activity data set and their interpretations by the CLP procedure. The table also lists for each variable its type (C for character, N for numeric), the possible values it can assume, and its default value.

Table 1.5 Activity Data Set Variables

Name	Type	Description	Allowed Values	Default
ACTIVITY	C/N	activity name		
DURATION	N	duration		0
SUCCESSOR	C/N	successor name	same type as _ACTIVITY_	
ALIGNDATE	N	alignment date		0
ALIGNTYPE	C	alignment type	SGE, SLE, SEQ, FGE, FLE, FEQ	SGE
LAG	C	lag type	FS, SS, FF, SF, FSE, SSE, FFE, SFE	FS
LAGDUR	N	lag duration		0

Schedule Data Set

In order to solve a scheduling-type CSP, you need to specify a schedule data set by using the SCHEDDATA= option in the PROC CLP statement. The Schedule data set contains all the solutions that have been determined by the CLP procedure.

The schedule data set always contains the following five variables: SOLUTION, ACTIVITY, DUR, START, and FINISH. If any resources have been specified, the data set also contains a variable corresponding to each resource specified in the RESOURCE statement having the same name as the resource. The SOLUTION variable gives the solution number that each observation corresponds to. The ACTIVITY variable identifies the activity. The DUR variable gives the duration of the activity. The START and FINISH variables give the scheduled start and finish times for the activity. If there are resources presented, then the corresponding resource variable indicates whether or not the resource is being used for the activity.

For every solution found and for each activity, the schedule data set contains an observation that lists the assignment information for that activity.

If an activity data set has been specified, then the formats and labels for the ACTIVITY and DUR variables carry over to the schedule data set.

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 requiring 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 requiring the same resource or set of resources. This in turn determines new time bounds on the start and finish times. Carlier and Pinson (1989) were responsible for some of the earliest work in this area that 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).

The edge-finding consistency routines are invoked 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 requiring 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 requiring 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 in 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 requiring the same resource (set of resources). The NOTFIRST= or NF= option in the SCHEDULE statement determines whether an activity is “not first.” In similar fashion, the NOTLAST= or NL= option in the SCHEDULE statement determines whether an activity is “not last.”

Macro Variable _ORCLP_

The CLP procedure defines a macro variable named _ORCLP_. This variable contains a character string that indicates the status of the procedure. It is set at procedure termination.

If the CLP procedure terminates successfully, the _ORCLP_ character string has one of the following formats:

- STATUS=SUCCESSFUL SOLUTION=INFEASIBLE
  PROC CLP successfully detected that the problem is infeasible.

- STATUS=SUCCESSFUL SOLUTIONS_FOUND=*n*
  PROC CLP found *n* solutions, where *n* can be zero.

- STATUS=SUCCESSFUL SOLUTION=TIMEOUT SOLUTIONS_FOUND=*n*
  PROC CLP successfully terminated due to the MAXTIME= parameter and found *n* solutions, where *n* can be zero.

If the CLP procedure terminates unsuccessfully, the form of the _ORCLP_ character string is STATUS=ERROR_EXIT REASON=*message*, where *message* can be one of the following:

- BADDATA_ERROR
- MEMORY_ERROR
- IO_ERROR
- SEMANTIC_ERROR
- SYNTAX_ERROR
- CLP_BUG
- UNKNOWN_ERROR

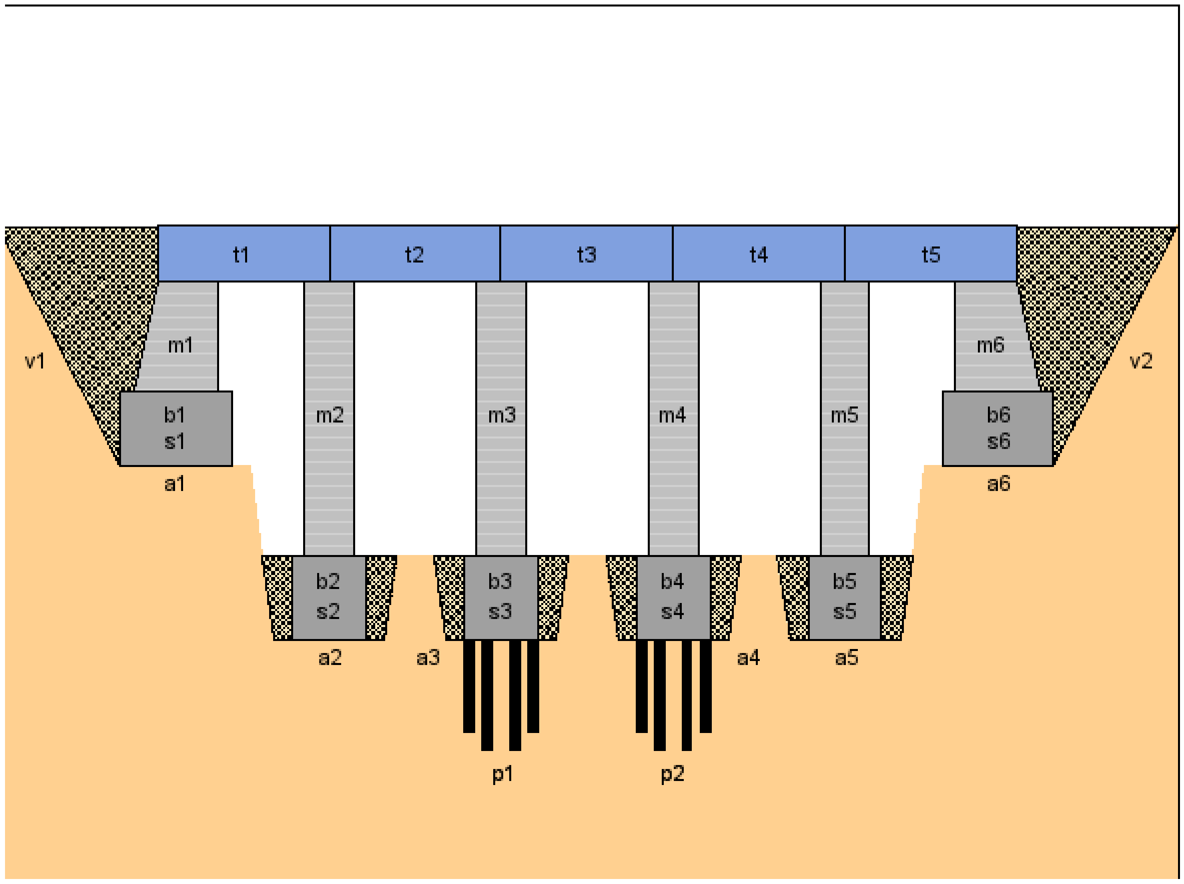
This information can be used when PROC CLP is one step in a larger program that needs to determine whether the procedure terminates successfully or not. Because _ORCLP_ is a standard SAS macro variable, it can be used in the ways that all macro variables can be used.

Examples: CLP Procedure

Example 1.1: Resource-Constrained Scheduling with Nonstandard Temporal Constraints

This example illustrates a real-life scheduling problem and is used as a benchmark problem in the CP community. The problem is to schedule the construction of a five-segment bridge (see Figure 1.1.1). It comes from a Ph.D. dissertation on scheduling problems (Bartusch 1983).

Output 1.1.1 The Bridge Problem



The project consists of 44 tasks and a set of constraints relating these tasks. Table 1.6 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 1.6 Data for Bridge Construction

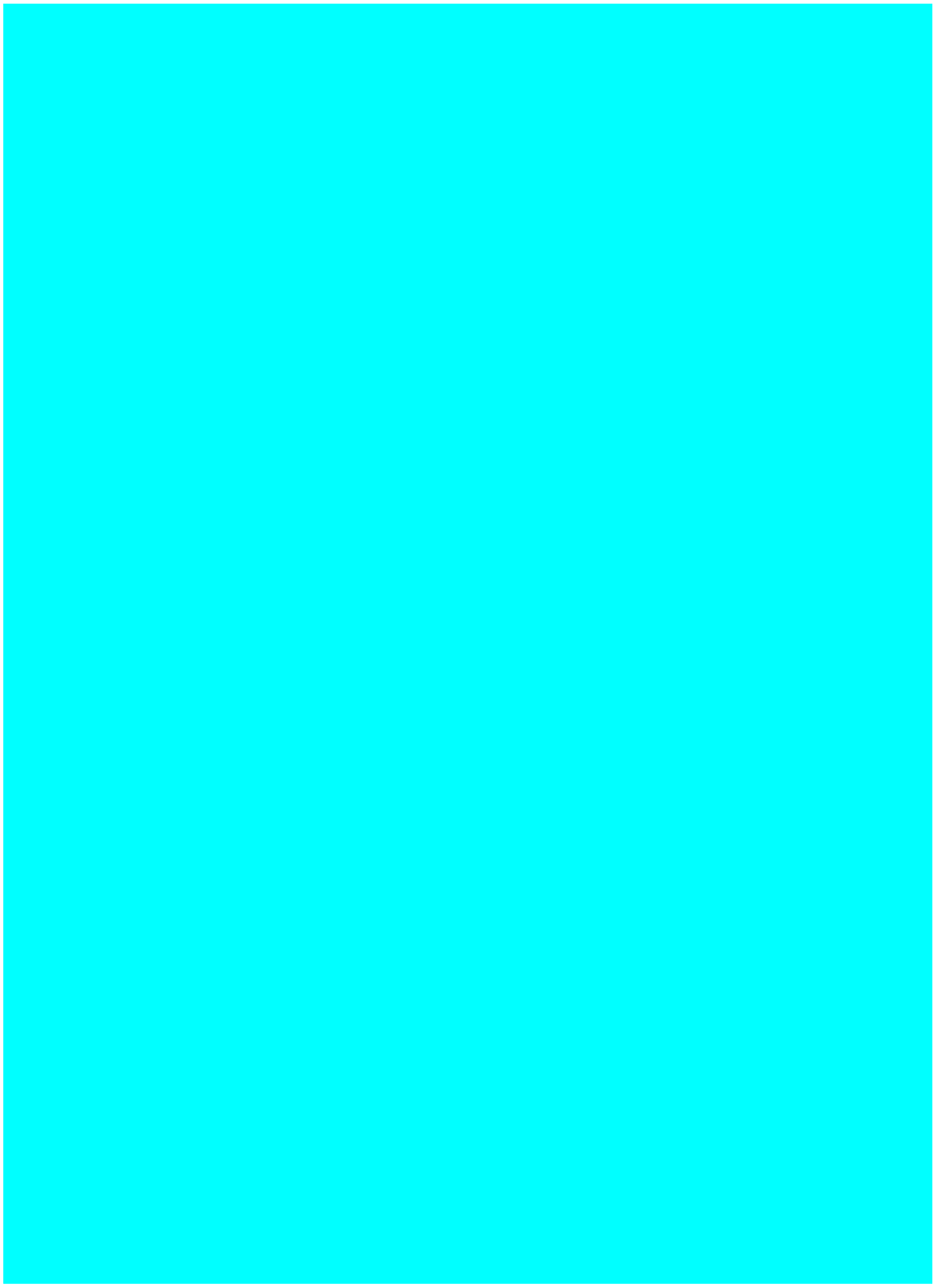
Activity	Description	Duration	Predecessors	Resource
pa	beginning of project	0		
a1	excavation (abutment 1)	4	pa	excavator
a2	excavation (pillar 1)	2	pa	excavator
a3	excavation (pillar 2)	2	pa	excavator
a4	excavation (pillar 3)	2	pa	excavator
a5	excavation (pillar 4)	2	pa	excavator
a6	excavation (abutment 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
s3	formwork (pillar 2)	4	p1	carpentry
s4	formwork (pillar 3)	4	p2	carpentry

Table 1.6 *continued*

Activity	Description	Duration	Predecessors	Resource
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	

A network diagram illustrating the precedence constraints in this problem is shown in Output 1.1.2. 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 NETDRAW procedure in SAS/OR software.

Output 1.1.2 Network Diagram for the Bridge Construction Project



In addition to the standard precedence constraints, there are the following constraints:

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

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

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

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

   and

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

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

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

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

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

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

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

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

The data set bridge defined by the SAS data set specification that follows, encapsulates all of the precedence constraints and also indicates the resources 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 1.5. The _RESOURCE_ variable is not used by the CLP procedure directly but used separately in a preprocessing step to generate the RESOURCE statement for the CLP procedure.

```
data bridge;
   format _ACTIVITY_ $32. _DESC_ $34. _RESOURCE_ $14.
          _SUCCESSOR_ $3. _LAG_ $3. ;
   input _ACTIVITY_ & _DESC_ & _DURATION_ _RESOURCE_ &
         _SUCCESSOR_ & _LAG_ & _LAGDUR_;
   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 SCHEDDATA= option, which is required for scheduling problems.

```
/* extract project descriptions */
proc sql;
   create table desc as
   select distinct _ACTIVITY_ as ACTIVITY,
                   _DESC_ as DESCRIPTION,
                   _RESOURCE_ as RESOURCE from bridge;
quit;

/* generate the REQUIRES statement */
data _null_;
   length reqspec $8192.;
   retain reqspec "";
   set desc end=final;
   if (RESOURCE ~= " ") then
      reqspec=
         trim(reqspec)||" "||trim(ACTIVITY)||"="||"("||trim(RESOURCE)||")";
   if final then
      call symput ('reqstmt', "REQUIRES"||reqspec);
run;

/* invoke PROC CLP */
proc clp actdata=bridge
         scheddata=sched_ex1
         dm=4
```

```
            solns=1
            restarts = 10
            showprogress;
      schedule edgefinder actselect=rand finish=104;
      resource excavator pile_driver carpentry concrete_mixer
               bricklaying crane caterpillar;
      &reqstmt;
   run;
```

The edge-finder consistency algorithms are activated using the EDGEFINDER option in the SCHEDULE statement. The REQUIRES statement listing all the resources is generated via the macro variable reqstmt. The FINISH= option is specified to find a solution in 104 days, which also happens to be optimal.

The sched_ex1 data set contains the complete schedule as computed by the CLP procedure, including the activity start and finish times and the assignment of resources. Since there is a variable for each resource in this data set and an activity gets assigned to at most one resource, it is possible to represent this information more concisely by merging the sched_ex1 data set with the desc data set by using the following statements. The resulting merged data set is shown in Output 1.1.3.

```
   /* merge descriptions, prepare to output the schedule data set */
   proc sort data=sched_ex1;
      by ACTIVITY;
   run;
   data sched_ex1;
      merge desc sched_ex1;
      by ACTIVITY;
   run;
   proc sort data=sched_ex1;
      by START FINISH;
   run;
      /* display the schedule */
   proc print data=sched_ex1 noobs width=min;
      var ACTIVITY DESCRIPTION START FINISH RESOURCE;
      title 'Bridge Construction Schedule';
   run;
```

Output 1.1.3 Bridge Construction Schedule

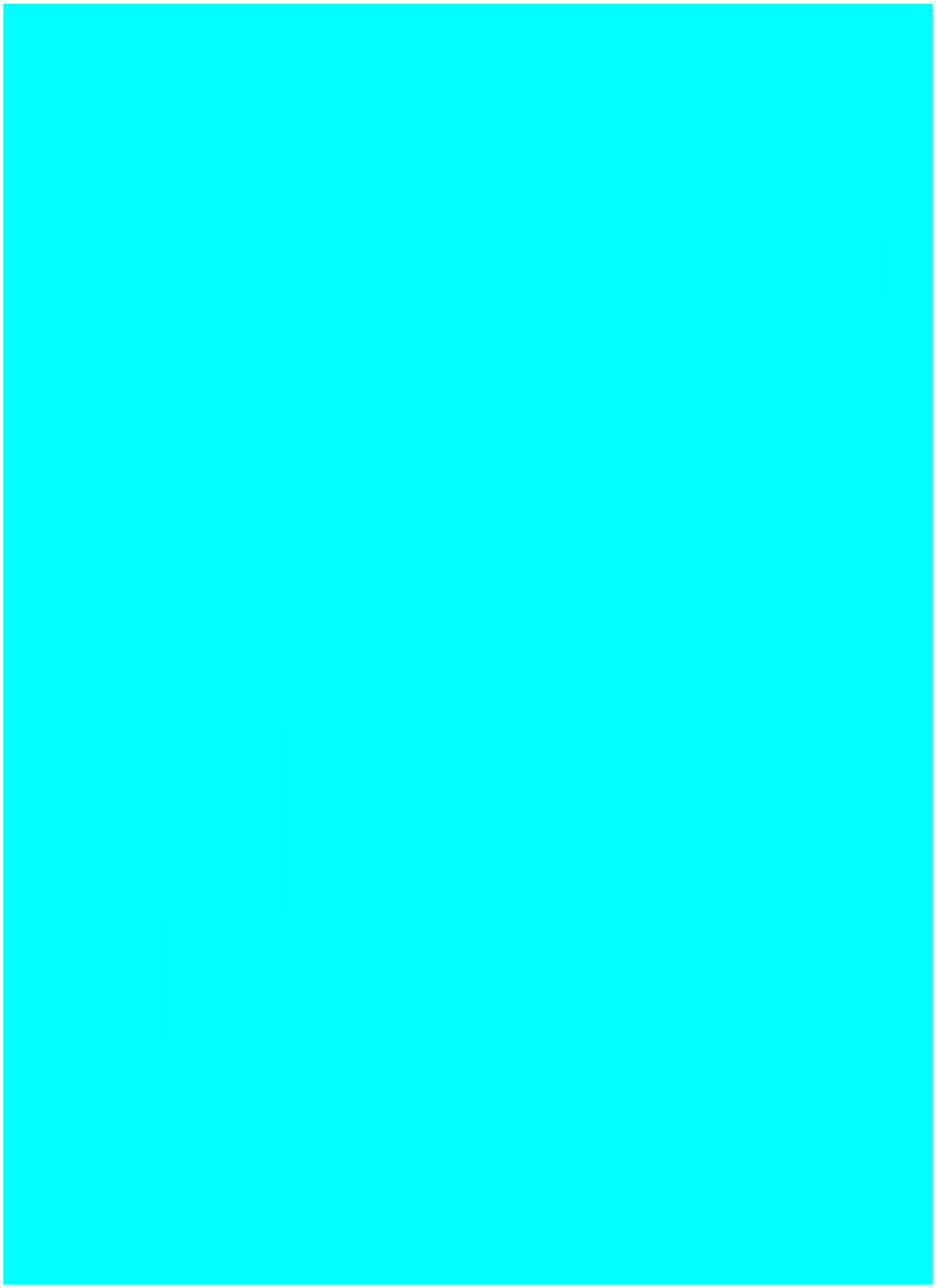
```
                            Bridge Construction Schedule

ACTIVITY    DESCRIPTION                            START    FINISH    RESOURCE

  pa        beginning of project                      0         0
  a4        excavation (pillar 3)                     0         2    excavator
  ue        erection of temporary housing             0        10
  a5        excavation (pillar 4)                     2         4    excavator
  p2        foundation piles 3                        2        15    pile_driver
  a1        excavation (abutment 1)                   4         8    excavator
  s5        formwork (pillar 4)                       6        10    carpentry
  a3        excavation (pillar 2)                     8        10    excavator
  b5        concrete foundation (pillar 4)           10        11    concrete_mixer
  s1        formwork (abutment 1)                    10        18    carpentry
  ab5       concrete setting time (pillar 4)         11        12
  m5        masonry work (pillar 4)                  12        20    bricklaying
  p1        foundation piles 2                       15        35    pile_driver
  a2        excavation (pillar 1)                    17        19    excavator
  b1        concrete foundation (abutment 1)         18        19    concrete_mixer
  s4        formwork (pillar 3)                      18        22    carpentry
  ab1       concrete setting time (abutment 1)       19        20
  a6        excavation (abutment 2)                  19        24    excavator
  m1        masonry work (abutment 1)                20        36    bricklaying
  b4        concrete foundation (pillar 3)           22        23    concrete_mixer
  s2        formwork (pillar 1)                      22        26    carpentry
  ab4       concrete setting time (pillar 3)         23        24
  b2        concrete foundation (pillar 1)           26        27    concrete_mixer
  s6        formwork (abutment 2)                    26        36    carpentry
  ab2       concrete setting time (pillar 1)         27        28
  l         delivery of the preformed bearers        30        32    crane
  b6        concrete foundation (abutment 2)         36        37    concrete_mixer
  s3        formwork (pillar 2)                      36        40    carpentry
  m2        masonry work (pillar 1)                  36        44    bricklaying
  ab6       concrete setting time (abutment 2)       37        38
  b3        concrete foundation (pillar 2)           40        41    concrete_mixer
  ab3       concrete setting time (pillar 2)         41        42
  m4        masonry work (pillar 3)                  44        52    bricklaying
  t1        positioning (preformed bearer 1)         44        56    crane
  m3        masonry work (pillar 2)                  52        60    bricklaying
  t4        positioning (preformed bearer 4)         56        68    crane
  v1        filling 1                                56        71    caterpillar
  m6        masonry work (abutment 2)                60        80    bricklaying
  t2        positioning (preformed bearer 2)         68        80    crane
  ua        removal of the temporary housing         78        88
  t5        positioning (preformed bearer 5)         80        92    crane
  v2        filling 2                                92       102    caterpillar
  t3        positioning (preformed bearer 3)         92       104    crane
  pe        end of project                          104       104
```

A Gantt chart of the schedule in Output 1.1.3, produced using the GANTT procedure in SAS/OR software, is displayed in Output 1.1.4. Each activity bar is also color coded according to the resource associated with it. The legend identifies the name of the resource associated with each color.

Output 1.1.4 Gantt Chart for the Bridge Construction Project



Example 1.2: Scheduling with Alternate Resources

This example shows an interesting job shop scheduling problem illustrating the use of alternative resources. There are 90 jobs (J1–J90) that need to be processed on one of 10 machines (M0–M9). Not every machine can process every job. In addition, certain jobs might also require one of 7 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 specified within the data set proj by using the following statements.

```
/* project specification */
data proj;
   array v{11} $8. j1-j3 m1-m3 o1-o4 dur;
   input v{*};
   datalines;
1    .    .    0    1    .    0    1    2    .    1
2    .    .    0    1    2    0    1    2    .    1
3    .    .    1    2    3    0    1    2    .    1
4    5    6    3    4    5    2    3    4    5    1
7    .    .    6    7    8    5    6    .    .    1
8    9    .    6    7    8    .    .    .    .    1
10   .    .    7    8    9    .    .    .    .    1
11   .    .    1    2    .    0    1    2    3    1
12   13   14   7    8    9    0    1    2    3    1
15   16   .    5    6    .    4    5    6    .    1
17   .    .    3    4    .    4    5    6    .    1
18   .    .    3    4    .    .    .    .    .    1
19   .    .    0    1    2    .    .    .    .    1
20   .    .    0    1    .    .    .    .    .    1
21   22   .    0    1    .    0    1    2    .    2
23   .    .    2    .    .    0    1    2    .    2
24   25   36   8    9    .    .    .    .    .    1
26   35   75   6    7    .    .    .    .    .    1
27   34   74   6    7    .    5    6    .    .    1
28   .    .    4    5    .    5    6    .    .    1
29   .    .    4    5    .    3    4    .    .    1
30   .    .    3    .    .    3    4    .    .    1
31   .    .    3    .    .    3    4    .    .    2
32   .    .    4    5    .    3    4    .    .    2
33   .    .    4    5    .    5    6    .    .    2
37   76   77   8    9    .    .    .    .    .    1
38   39   .    0    1    .    0    1    .    .    1
40   .    .    2    .    .    2    .    .    .    1
41   .    .    6    7    .    6    .    .    .    2
42   62   82   6    7    .    .    .    .    .    2
43   44   63   8    9    .    .    .    .    .    2
45   46   65   0    1    .    4    5    .    .    2
47   67   .    2    3    .    2    3    .    .    2
48   68   .    2    3    .    2    3    .    .    1
49   50   69   4    5    .    0    1    .    .    1
```

```
51   52   .    3    4    .    1    2    .    .    2
53   .    .    5    6    .    0    .    .    .    2
54   .    .    5    6    .    6    .    .    .    1
55   56   57   7    8    9    .    .    .    .    1
58   59   78   0    1    2    3    4    .    .    1
60   80   .    0    1    2    5    6    .    .    1
61   81   .    6    7    .    5    6    .    .    2
64   83   84   8    9    .    .    .    .    .    2
66   .    .    0    1    .    4    5    .    .    2
70   .    .    5    .    .    0    1    .    .    1
71   .    .    5    .    .    0    1    2    .    2
72   73   .    3    4    .    0    1    2    .    2
79   .    .    0    1    2    3    4    .    .    1
85   .    .    0    .    .    3    .    .    .    1
86   .    .    1    .    .    4    .    .    .    1
87   .    .    2    .    .    5    .    .    .    1
88   .    .    3    .    .    0    .    .    .    1
89   .    .    4    .    .    1    .    .    .    1
90   .    .    5    .    .    2    .    .    .    1
;
```

Each row in the datalines section defines a resource requirement for up to three jobs that are identified in the variables j1–j3. The resource requirement associated with each of these jobs is a set of alternates and is defined using the variables m1–m3 and o1–o4.

The variables m1–m3 represent a machine or workstation ID that the job needs to be processed on, and the variables o1–o4 represent an operator ID. The duration of each of the jobs identified in j1–j3 is assumed to be identical and defined using the dur variable. Each of the jobs in a row can be processed on any one of the machines in m1–m3 and requires the assistance of one of the operators in o1–o4 while being processed. In other words, the resource requirement for each job is a conjunction of disjunctive requirements (or alternates). A job requires one of m1–m3 and one of o1–o4 in order to be processed. For example, observation 5 specifies that job number 7 can be processed on either machine 6, 7, or 8 and additionally requires either operator 5 or operator 6 in order to be processed. The next observation indicates that jobs 8 and 9 can also be processed on the same set of machines. However, they do not require any operator assistance.

The ACTIVITY and REQUIRES statements for the CLP procedure are next generated from the data set proj by the following program:

```
/* generate the ACTIVITY and REQUIRES statements */
data _null_;
   set proj end=finish;
   format jobs $32. resources $char128.;
   format acts reqs $char4096.;
   retain acts reqs;
   array v{11} j1-j3 m1-m3 o1-o4 dur;
   jobs=catx(' J',of j1-j3);
   acts=catx(' ',acts,cats('(J',jobs,')=','(',dur,')'));
   do i=4 to 6;
      if v{i}=' ' then leave;
      if v{7}=' ' then resources=catx(',',resources,'M'||v{i});
      else do j=7 to 10;
         if v{j}=' ' then leave;
         resources=catx(',',resources,catx(' ','M'||v{i},'OP'||v{j}));
         end;
   end;
   reqs=catx(' ',reqs,cats('(J',jobs,')=','(',resources,')'));
   if finish then do;
   call symput('activities',strip(acts));
   call symput('requirements',strip(reqs));
   end;
run;
```

The CLP procedure is invoked by using the following statements with DUR=12 to obtain a 12-day solution that is also known to be optimal.

```
/* invoke PROC CLP to find a schedule */
proc clp dom=[0,12] scheddata=sched_ex2 restarts=500 dpr=25;
   schedule start=0 dur=12 actselect=MAXD actassign=MAXTW EDGEFINDER;
   activity &activities;
   resource (M0-M9) (OP0-OP6);
   requires &requirements;
run;
```

The activity selection strategy is one that selects a maximum duration activity at random from the subset of activities that begin prior to the earliest early finish time. This strategy is specified using the ACTSELECT=MAXD option on the SCHEDULE statement.

The resulting schedule is shown in a series of Gantt charts that are displayed in Output 1.2.1 and Output 1.2.2. 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 1.2.1 displays the schedule for the operator-assisted tasks—one for each operator. Output 1.2.2 shows the schedule for automated tasks—that is, those tasks not requiring operator intervention.

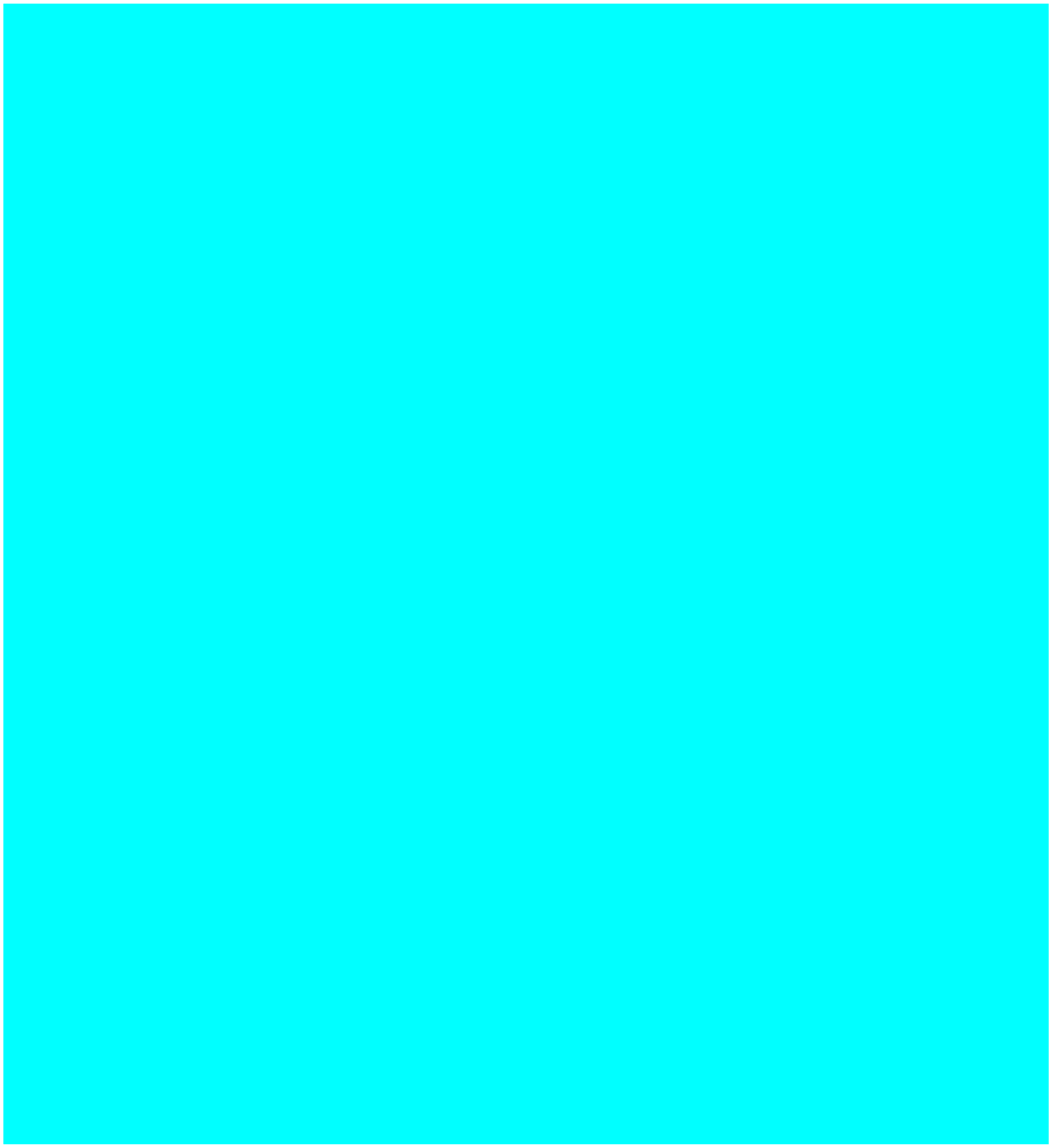
Output 1.2.1 Operator-Assisted Jobs Schedule

Output 1.2.1 *continued*

Output 1.2.1 *continued*

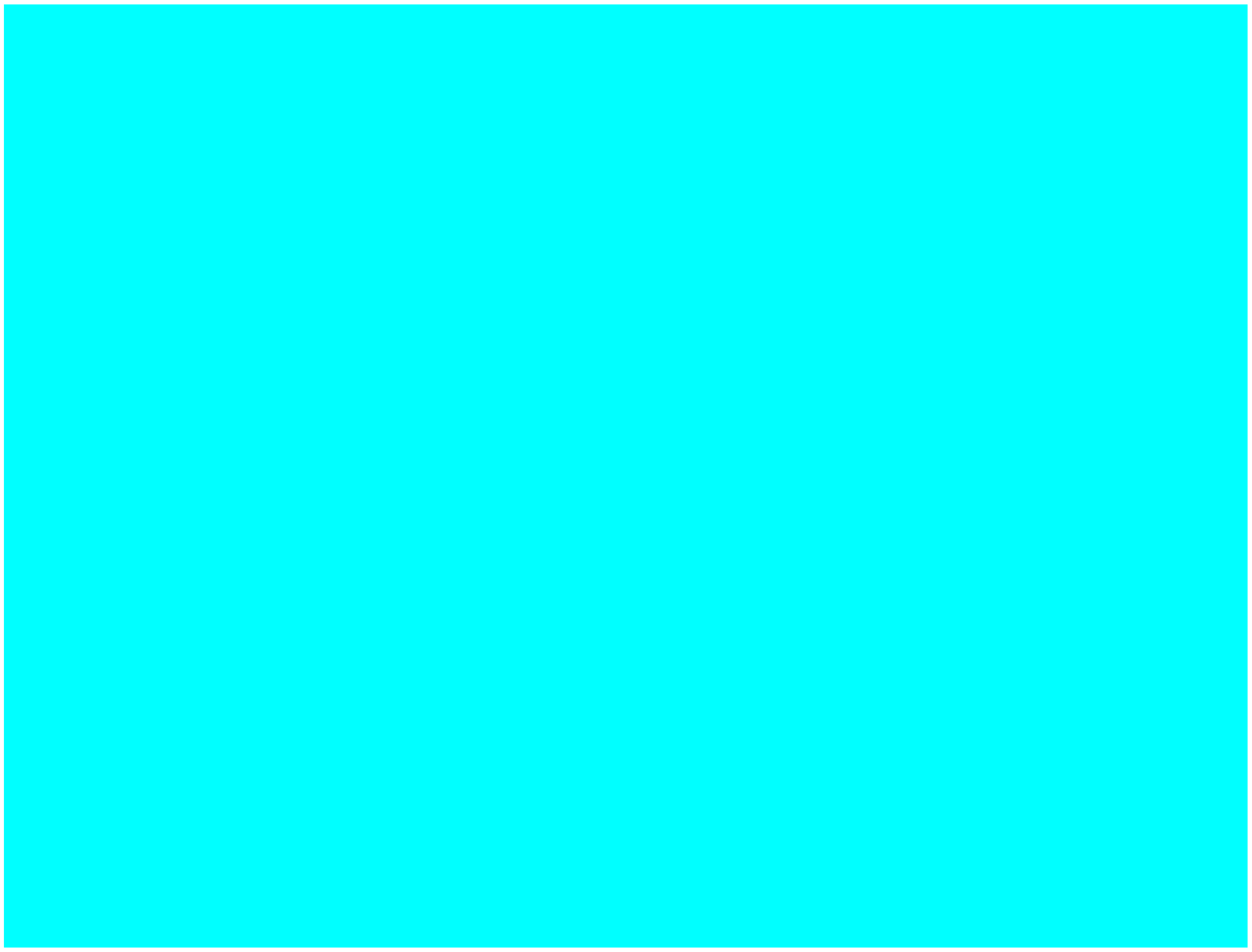
Output 1.2.1 *continued*

Output 1.2.2 Automated Jobs Schedule



A more interesting Gantt chart is that of the resource schedule by machine, as shown in Output 1.2.3. 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 1.2.3 Another Gantt Chart: Proof of Optimality



Example 1.3: 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 solution is known to be 842 (Applegate and Cook 1991). There are 10 jobs (Job 1 – 10) 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 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_ @;
      MACHINE=compress('M'||mid);
      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
;
```

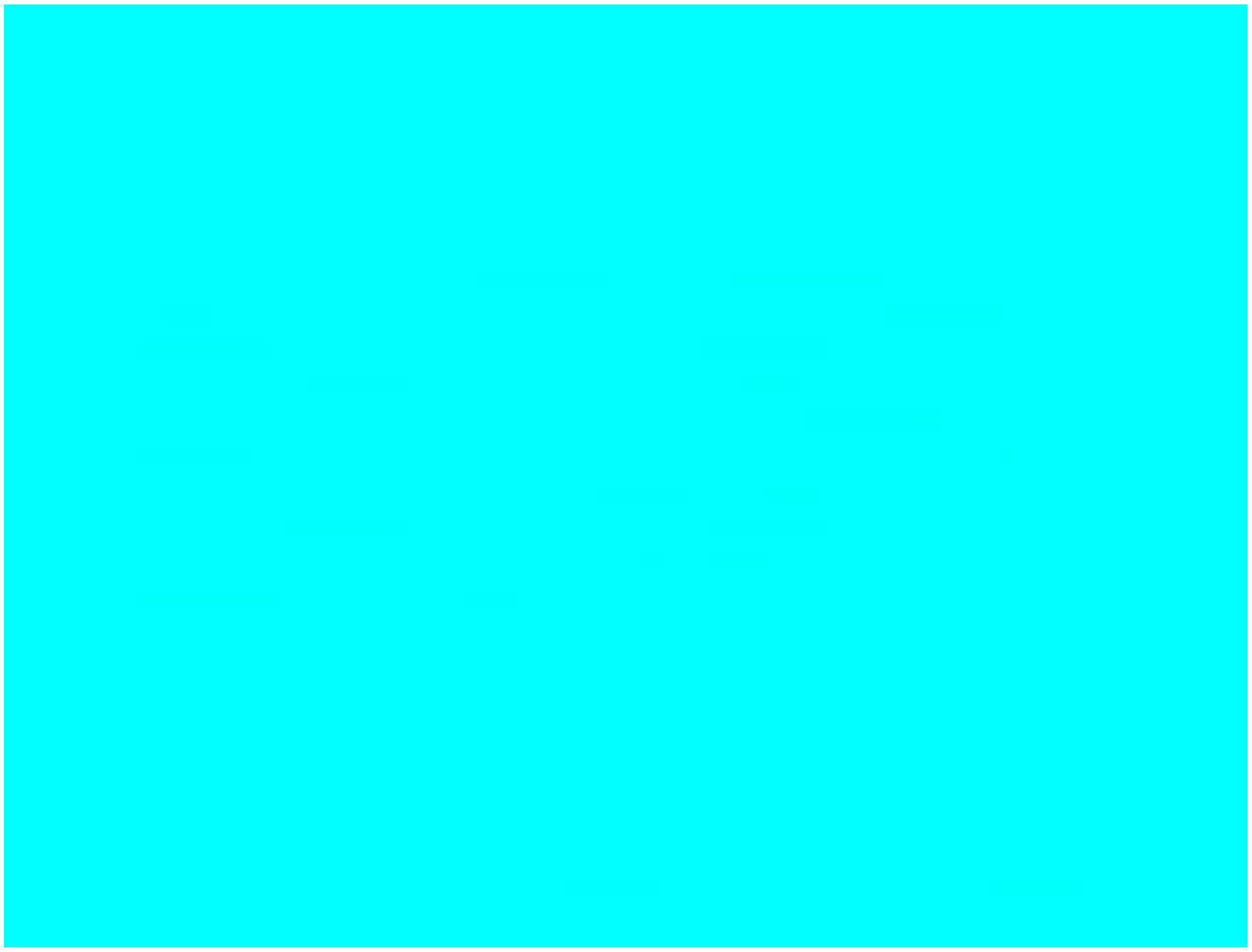
Each row in the datalines section specifies a job by 10 pairs of consecutive numbers. Each pair of numbers defines one task of the job, which represents the processing of a job on a machine. For each pair, the first number identifies the machine it executes on and the second number is the duration. The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

The following statements create the activity data set actdata.

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

Had there been sufficient machine capacity, the jobs could have been processed according to a schedule as shown in Output 1.3.1. The minimum makespan would be 617—the times it takes to complete Job 1.

Output 1.3.1 Gantt Chart: Schedule for the Unconstrained Problem



This schedule will be 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.

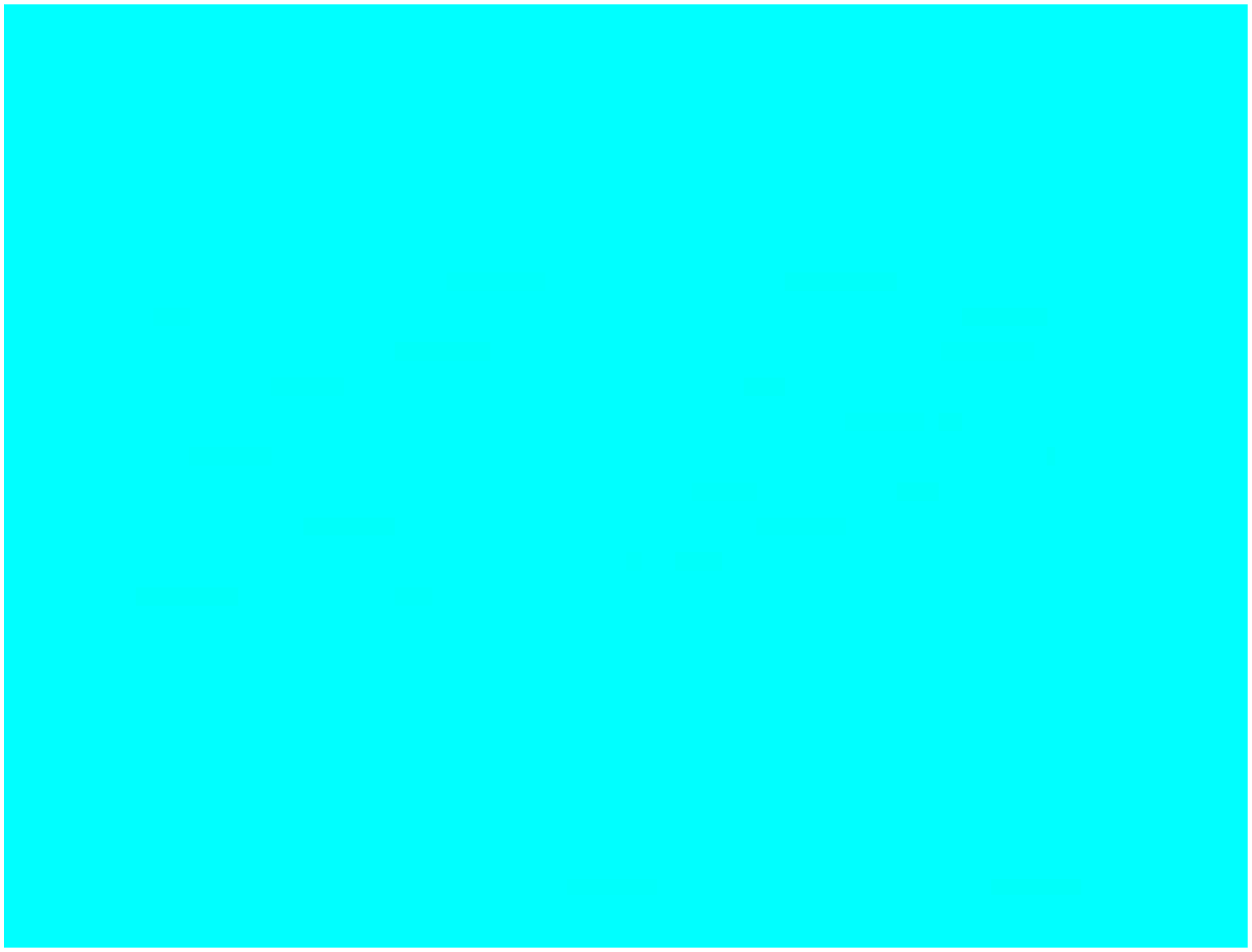
```
/* generate the REQUIRES statement */
data _null_;
   length reqspec $1100.;
   retain reqspec '';
   set actdata end=finish;
   reqspec=catx(' ', reqspec, cats(_ACTIVITY_,'=(',MACHINE,')'));
   if finish then
      call symput('reqstmt', catx(' ', 'REQUIRES ', reqspec));
run;

/* invoke PROC CLP to find a resource-constrained schedule */
proc clp domain=[0,842]
         actdata=actdata
         scheddata=sched_ex3
         dpr=50
         restarts=500
         showprogress;
   schedule dur=842 edgefinder NF=1 NL=1;
   resource (M0-M9);
   &reqstmt;
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 from a subset of jobs 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 500. The REQUIRES statement for the CLP procedure is generated via the macro variable reqstmt.

The resulting 842-time-period schedule is displayed in Output 1.3.2. 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 1.3.2 Gantt Chart: Optimal Resource-Constrained Schedule



References

Applegate, D. 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 15th Workshop of the UK Planning Special Interest Group*, Liverpool, UK.

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

Carlier, J. and Pinson, E. (1989), "An Algorithm for Solving the Job-Shop Scheduling Problem," *Management Science*, 35(2), 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.

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

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

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

Jaffar, J. and Lassez, J. (1987), “Constraint Logic Programming,” *Conference Record of the 14th Annual ACM Symposium in Principles of Programming Languages*, Munich, 111–119.

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)*, Pittsburgh: Graduate School of Industrial Administration, Carnegie Mellon University.

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

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

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. thesis, Eindhoven Institute of Technology.

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.

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., Deville, Y., and Teng, C. (1992), “A Generic Arc-Consistency Algorithm and Its Specializations,” *Artificial Intelligence*, 57, 291–321.

Waltz, D. L. (1975), “Understanding Line Drawings of Scenes with Shadows,” in P. H. Winston, ed., *The Psychology of Computer Vision*, 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 of Computing*, 10, 261–264.